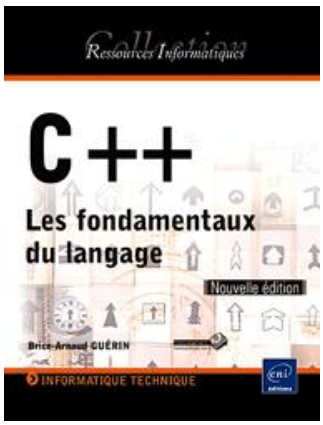
C++

Les fondamentaux du langage [Nouvelle édition]

Brice-Arnaud GUÉRIN



Résumé

Ce livre s'adresse à tout **développeur** désireux d'apprendre le langage C++, dans le **cadre de ses études** ou pour **consolider son expérience** professionnelle.
Le premier chapitre présente les bases de la **syntaxe** du langage ainsi que l'organisation des programmes. Le chapitre suivant est une transition vers C++, il explicite les **notions clés** pour créer ses premières applications : **structures, pointeurs**, **bibliothèques standard**... Le troisième chapitre détaille la **programmation orientée objets** et les mécanismes spécifiques au langage (héritage, modèles de classes...). Vient ensuite l'étude de la **STL** (**Standard Template Library**), présentée à travers ses mécanismes les plus importants : les **chaînes**, les **structures de données** et les **algorithmes.** Le chapitre 5 ouvre C++ sur ses univers, le framework **MFC** et l'environnement **.NET C++ CLI**.
Comme illustration des capacités de C++ à créer tout type d'applications, l'ouvrage propose un **exemple complet de tableur graphique** ou encore un **grapheur 3D**. L'ouvrage se termine par un chapitre consacré à l'**optimisation** et aux méthodes de conception orientée objet (**UML**).
Le code source des exemples du livre est disponible en téléchargement sur www.editions-eni.fr.

Les chapitres du livre :
Avant-propos - Introduction - De C à C++ - Programmation orientée objet - La bibliothèque Standard Template Library - Les univers de C++ - Des programmes C++ efficaces

L'auteur

Ingénieur ESIEA, **Brice-Arnaud GUÉRIN** est responsable des développements logiciels chez LexisNexis, acteur majeur dans l'information juridique, économique et financière, et les solutions de gestion pour les professionnels. Ses compétences en développement, son désir de partager ses connaissances l'ont naturellement conduit à l'écriture d'ouvrages consacrés à la réalisation d'applications (.NET, PHP, C++) et à la conduite de projets.

Objectifs de ce livre

C++ fait figure de référence au panthéon des langages informatiques, ce qui est tout à fait justifié de par sa très large application. Il est également réputé difficile et réservé à des connaisseurs, voire à des experts. Cette appréciation est pourtant infondée.

L'objectif premier de ce livre est d'expliquer les fondamentaux du langage en s'appuyant sur « la petite histoire » qui a conduit son concepteur, Bjarne Stroustrup, à définir C++ tel que nous le pratiquons aujourd'hui. Le chapitre Introduction présente les bases de la syntaxe ainsi que l'organisation des programmes, éléments qui n'ont pas cessé d'évoluer depuis l'introduction du langage. Le chapitre De C à C++ accompagne le lecteur jusqu'aux portes de C++, en passant par l'étape langage C. Cela permet d'aborder en douceur ces notions clés que sont les pointeurs, les structures et les types de données définis dans les bibliothèques standard. Il devient alors plus aisé d'appréhender les concepts du chapitre Programmation orientée objet ainsi que leurs applications.

L'angle pratique est le deuxième objectif de ce livre, car les mécanismes les plus abscons ne se révèlent qu'au travers d'exemples concrets et aussi peu artificiels que possible. Le chapitre La bibliothèque Standard Template Library présente la bibliothèque standard STL sous la forme d'un condensé des fonctions les plus incontournables. Sans être un guide de référence, cela donnera au développeur les points d'entrée pour bâtir ses premières applications. Le chapitre Les univers de C++ propose plusieurs applications « d'envergure » comme le grapheur 3D ou le tableur pour Windows. Avoir un but conséquent en programmation favorise le dépassement du simple niveau syntaxique, et la découverte de frameworks tels que MFC ou .NET donnera un éclairage indispensable pour envisager le développement de logiciels écrits en C++.

Finalement, C++ s'avère être un langage à la fois concis et expressif. Le nombre de constructions est potentiellement infini mais les éléments de base sont réduits à une poignée de mots-clés et à quelques dizaines de règles de grammaire. Syntaxe contenue mais sémantique sans limite (comme XML en fait), l'enjeu du programmeur C++ est évidemment de maîtriser la complexité et de conserver « la qualité » au fil des cycles de développement. Le chapitre Des programmes C++ efficaces propose des pistes pour coder avec efficacité (c'était justement ce à quoi Bjarne Stroustrup voulait arriver), mais aussi concevoir des réseaux de classes qui n'enfouiraient pas l'essentiel derrière des millions de lignes de code. Au-delà de la STL et de son approche pragmatique, c'est bien UML et les méthodes d'analyse basées sur cette notation qu'il faut considérer.

À qui s’adresse ce livre ?

Ce livre est destiné à tout développeur désireux d’apprendre le langage C++, dans le cadre de ses études, ou pour consolider une expérience professionnelle.

Avant d’aborder cet ouvrage, la maîtrise d’un autre langage de programmation est un plus, mais il n’est pas nécessaire que ce langage soit apparenté au langage C.

Tous les exemples ont été réalisés avec la version gratuite de Visual C++, et une très grande majorité d’entre eux sont portables tels quels sous Unix et Linux, à l’exception des exemples du chapitre Les univers de C++ qui pourront être testés avec quelques aménagements.

Les notions clés

1. Principales caractéristiques du langage C++

Le langage C++ est apparu officiellement en 1983, date de ses premières utilisations hors du laboratoire AT&T qui l'a fait naître. Son concepteur, Bjarne Stroustrup, avait débuté ses travaux plusieurs années auparavant, sans doute vers 1980. Le langage C++ - appelé jusque-là C avec des classes - a été élaboré en conservant la plupart des concepts du langage C, son prédécesseur. Les deux langages se sont ensuite mutuellement fait des emprunts.

Comme le langage C, C++ adopte une vision très proche de la machine. Il a été destiné en premier lieu à l'écriture de systèmes d'exploitation mais ses caractéristiques lui ont ouvert d'autres perspectives.

Le langage est formé d'instructions très explicites, courtes, dont la durée d'exécution peut être prévue à l'avance, au moment de l'écriture du programme.

Le nombre d'instructions et de notations étant volontairement limité, les interprétations des constructions sémantiques sont multiples et c'est sans doute ce que le concepteur du langage C++ désigne sous le terme d'expressivité.

Toutefois, Bjarne Stroustrup a veillé à contourner certains écueils du langage C, et notamment sa tendance à tout ramener au niveau de l'octet, quantité numérique limitée et totalement obsolète dans l'histoire de l'informatique, même en 1980. Pour obtenir ce résultat, le langage s'est enrichi de classes qui décrivent des types de données adaptés aux différents besoins du programmeur. La visibilité du langage combinée à l'abstraction des classes fournit des programmes de haut niveau.

Les classes (et la programmation orientée objet) n'ont pas été inventées à l'occasion de C++. L'inventeur du langage s'est efforcé d'adapter des concepts de haut niveau introduits par le langage Simula en 1967, à une plate-forme très explicite, le langage C.

Il résulte de cette double filiation un langage très riche, très puissant, particulièrement expressif et en même temps très efficace. Le langage s'est ensuite affiné, a subi des évolutions, des transformations, jusqu'au stade où le produit de laboratoire est devenu un standard, labellisé par l'organisme américain ANSI. Les travaux de standardisation ont débuté en 1987 pour s'achever en 1998.

Sans attendre cette date tardive, C++ avait fait de nombreux adeptes. En premier lieu, les programmeurs connaissant le langage C sont passés plus ou moins naturellement au langage C++ : nombre de compilateurs C++ reconnaissent le langage C. Et de nombreux éditeurs de compilateurs ne proposent pas de compilateur C seul. En second lieu, le langage a servi à l'écriture de systèmes d'exploitation ou de parties de systèmes d'exploitation : le système équipant le Macintosh, l'interface graphique Windows sont codés en C++. Par conséquent, les logiciels prévus pour ces systèmes avaient tout avantage à être programmés eux aussi en C++.

En résumé, on pourrait dire que C++ est tout à la fois un langage de haut niveau basé sur des instructions proches de la machine. Un équilibre subtil que les développeurs apprécient.

2. Programmation orientée objet

L'emploi généralisé de méthodes de conception logicielles telles qu'UML a eu raison de l'histoire de l'informatique. Pourquoi la programmation orientée objet a-t-elle été inventée ? Que permet-elle de décrire en particulier ? Nous devons nous pencher sur le sujet car une des caractéristiques les plus importantes du langage C++ est son goût prononcé pour la production de classes et d'objets.

Vers 1950, les ordinateurs étaient programmés à l'aide du langage Assembleur. Ce langage, de très bas niveau puisqu'il nécessite la fourniture de codes binaires, respecte rigoureusement les consignes de Von Neumann, l'inventeur des ordinateurs séquentiels. Un programme est constitué d'une suite d'instructions impératives, que le processeur va exécuter les unes à la suite des autres, ainsi que de données que l'on n'ose pas encore appeler "variables". Tout est donc clair, un programme possède un début et une fin. Il consomme des entrées et fournit un résultat lorsque s'achève son exécution.

Les microprocesseurs sur circuit intégré n'existaient pas encore. Les premiers circuits intégrés sont apparus à la fin des années soixante et les premiers microprocesseurs intégrés ont été mis sur le marché au début des années soixante-dix. Jusque-là, les ordinateurs étaient conçus autour d'un processeur à base de composants discrets, transistors ou lampes.

On s'est rapidement rendu compte que cette façon d'aborder les programmes posait deux problèmes. D'une part, le programme était loin d'être optimal car de nombreuses suites d'instructions se répétaient, pratiquement à l'identique. D'autre part, les programmes étaient élaborés par des mathématiciens qui cherchaient à traduire des algorithmes faisant appel à des outils conceptuels de bien plus haut niveau que l'Assembleur.

C'est en 1954 que les travaux sur le langage Fortran (*Formula Translator*) ont débuté. En 1957, John Backus présentait un compilateur destiné aux scientifiques qui avaient besoin de programmer des algorithmes structurés sans avoir à construire eux-mêmes la traduction en Assembleur. D'autres langages ont suivi reprenant le modèle de Fortran : des instructions de contrôle (if, for, while...), des variables, et finalement des procédures.

Une procédure est une suite d'instructions qui porte un nom. Le pas franchi était énorme : plutôt que de recréer à l'identique une suite d'instructions, on place la première occurrence dans une procédure, puis on remplace les prochaines occurrences par un appel à cette procédure. La programmation procédurale était née, bientôt appelée programmation fonctionnelle, ce qui permettait de relier mathématiques et informatique.

De nos jours, presque tous les langages connaissent la notion de procédure : Basic, Pascal, Java, et bien entendu C et C++.

La prochaine étape consistait à structurer les données. En étudiant de près un programme qui manipule des coordonnées de points dans le plan, on s'aperçoit que chaque point est défini par deux variables x et y, l'une représentant l'abscisse du point et l'autre, son ordonnée. Les langages de programmation des années 50 s'accommodaient mal de ce type de situation. Pour représenter trois points P1, P2, P3 il fallait déclarer six variables, x1, y1, x2, y2, x3, y3. La programmation structurée (à base de structures) autorise la définition de types rassemblant un certain nombre de variables aussitôt dénommées champs. Dans notre exemple, nous imaginons le type Point contenant les champs x et y. L'intérêt de cette approche réside dans le fait que le nombre de variables est divisé par deux (trois instances de Point : P1, P2 et P3). Beaucoup de langages structurés ont adopté la notation P1.x pour désigner la valeur du champ x rapportée au point P1, autrement dit, l'abscisse de P1.

Les programmes bâtis avec ces langages décrivent des types structurés, des variables et des procédures (fonctions).

Une fonction est une procédure qui prend généralement des paramètres pour évaluer un résultat. Après avoir mis au point la programmation structurée, on s'est rendu compte que les fonctions prenaient le plus souvent des instances de structures comme paramètres. La syntaxe des langages de programmation s'alourdissait rapidement, puisqu'il fallait distinguer de nombreux cas de figure : passage par valeur, par référence, structure en entrée, en sortie...

Vint alors l'idée de réunir dans une seule "boîte" la structure, des champs et des fonctions s'appliquant à ces champs. Cette structure devenait une classe. Les variables formées à partir d'une classe (les instances) prenaient le nom d'objet. C'était à n'en point douter la voie à suivre, l'avenir. La programmation orientée objet était née, et le langage Simula, proposé en 1967 fut une de ses premières concrétisations.

Reste que ce type de programmation n'a pas connu à cette époque l'engouement qu'on lui reconnaît de nos jours. Les concepts étaient peut-être trop novateurs, trop ardus pour l'informatique naissante. De plus, les ordinateurs de l'époque, à peine transistorisés, avaient le plus grand mal à faire fonctionner cette programmation gourmande en octets.

On se concentra donc sur la mise au point de langages destinés à l'efficacité, on "descendit" dans l'échelle de la conceptualisation pour revenir à des valeurs sûres. Le langage C est vraisemblablement un des langages les plus explicites qui soit, toujours prêt à revenir à l'unité fondamentale, l'octet.

Une quinzaine d'années s'écoulèrent et Bjarne Stroustrup perçut le retard pris par de nombreux projets informatiques, faute d'outils logiciels adaptés. La plupart étaient de très bas niveau, proches de l'Assembleur, alors que les problèmes à traiter devenaient de plus en plus conceptuels. Heureusement, le hardware avait fait de gros progrès depuis 1967, avec l'introduction des circuits intégrés puis des microprocesseurs, la mémoire vive coûtait de moins en moins cher et même les disques durs étaient livrés en série avec les ordinateurs personnels.

Ces conditions favorables réunies, Bjarne Stroustrup opéra la greffe des travaux Simula sur un compilateur C ; le langage C++ tire ses origines de cette approche.

3. Environnement de développement, makefile

a. Choix d'un EDI

La donne a un peu changé ces dernières années. Auparavant, l'environnement de développement intégré (EDI) proposait son éditeur de code source, son compilateur, ses librairies et ses utilitaires. Il s'agissait d'un logiciel propriétaire qui évoluait environ une fois par an.

On devrait désormais parler d'environnement de développement intégré. L'introduction de la technologie XML a rendu les fichiers de configuration plus ouverts. Les éditeurs de code source sont personnalisables, le compilateur peut être remplacé par un autre, tout est modifiable. Le paroxysme est atteint avec Eclipse qui devient une plate-forme de développement pour pratiquement tous les langages existants, par le biais de modules additionnels (plug-ins).

C++ Builder	Embarcadero (repreneur des activités développement de Borland) Licence commerciale	Le compilateur seul en ligne de commande est gratuit, et l'EDI est payant.
Visual C++	Microsoft Licence commerciale mais édition express gratuite.	Offre un éditeur visuel d'applications graphiques, plus les framework MFC et .NET. Une version "libre" est disponible. Des plug-ins sont disponibles, notamment pour utiliser le compilateur Intel.
Dev C++	Open Source	Intègre le compilateur GNU. Pas tellement adapté au développement graphique pour Windows.
CygWin	Open Source	Environnement Unix pour Windows.
Eclipse	Open Source	Plate-forme d'édition de code source, initialement destinée à Java. Des plug-ins C++ sont disponibles.
Open Watcom	Libre	Plate-forme de développement croisé Watcom.

Cette liste est à nouveau loin d'être exhaustive. De nombreux EDI offrent un grand confort à l'édition de code source. La disponibilité d'un débuggeur, une documentation en ligne de qualité seront des éléments qui vous aideront à choisir. Des magazines de programmation publient régulièrement leur avis sur ce type de produit, test à l'appui.

b. Construction d'un fichier makefile

Le fichier **`makefile`** spécifie les opérations de construction d'un programme ; il est indépendant du langage de programmation et s'applique très bien à C++. Certains environnements de développement l'ont remplacé par leur propre gestion de "projets", avec parfois la possibilité de convertir un `makefile` en projet ou d'exporter un projet en `makefile`.

Il n'existe en principe qu'un seul fichier `makefile` par répertoire. L'utilitaire qui pilote la compilation - **`make`** ou **`nmake`** - peut être lancé en spécifiant le fichier à utiliser, mais `makefile` est celui choisi par défaut.

Un fichier `makefile` se construit à l'aide d'un éditeur de texte. Il contient un ensemble de cibles. On rencontre le plus fréquemment les cibles all et clean.

La syntaxe est la suivante :

```
cible : dépendance1 dépendance2 ...
     instruction1
     instruction2
     instruction3
```

Les dépendances et les instructions sont optionnelles. Souvent, la cible all n'a pas d'instruction, et la cible clean n'a pas de dépendance. Voici un petit exemple de `makefile` :

```
#cibles par défaut
all : lecteur.exe

lecteur.exe : lecteur.obj personne.obj
    link lecteur.obj personne.obj -o lecteur.exe

lecteur.obj : lecteur.cpp lecteur.h personne.h
    cl -c lecteur.cpp -o lecteur.obj

personne.obj : personne.h personne.cpp
    cl -c personne.cpp -o personne.obj


#nettoyage du répertoire
clean :
    del *.obj
    del lecteur.exe
```

Donnons une lecture en langue française de ce fichier `makefile` et précisons tout d'abord que les lignes commençant par un signe # sont des commentaires, ainsi qu'il en est l'usage dans les fichiers script Unix.

Le fichier `makefile` commence par identifier la cible `all` qui n'a que des dépendances. Cette syntaxe indique qu'il faut construire le programme `lecteur.exe`. Lorsque l'on exécute la commande `make` depuis le répertoire du projet, les dates des fichiers sont comparées en remontant les règles les unes après les autres :

- si **`personne.h`** ou **`personne.cpp`** ont une date postérieure à **`personne.obj`**, le fichier **`personne.cpp`** est recompilé ;
- si **`lecteur.cpp`**, **`lecteur.h`** ou **`personne.h`** ont une date postérieure à **`lecteur.obj`**, le fichier **`lecteur.cpp`** est aussi recompilé ;
- si l'un des modules objets a une date postérieure au fichier **`lecteur.exe`**, la commande **`link`** est elle aussi exécutée.

Si l'un des modules objets a une date postérieure au fichier **`lecteur.exe`**, la commande **`link`** est elle aussi exécutée.

Ce processus s'appelle la compilation séparée ; les modules ne sont recompilés que si c'est nécessaire. Sous Unix, on dispose d'une commande, **`touch`**, qui modifie la date de certains fichiers. Cela donne le même effet que d'ouvrir lesdits fichiers dans un éditeur et de les réenregistrer sans modification.

Toutefois, afin de s'assurer de la fraîcheur des résultats, la commande **`make clean`** efface tous les modules objets, ainsi que l'exécutable.

On peut alors enchaîner les commandes **`make clean`** et **`make`**. Les fichiers intermédiaires ayant été effacés, tous les modules sont recompilés.

Une syntaxe de macros facilite le changement de compilateur. On définit un certain nombre de constantes, telles que le nom du compilateur, le nom de l'éditeur de liens, les commutateurs d'optimisation, la liste des modules objets...

4. Organisation d'un programme C++

Un programme C++ subit une série de transformations avant d'être exécuté. La principale transformation s'effectue à l'aide d'un outil que l'on appelle compilateur. Le compilateur fait lui-même appel à un prétraitement confié au préprocesseur. Le préprocesseur rassemble différents fichiers contenant des sources à compiler ensemble.

Le résultat de la compilation est placé dans un ou plusieurs modules objets, qui sont ensuite réunis par un troisième outil appelé éditeur de liens.

L'éditeur de liens forme généralement un exécutable pour le système d'exploitation, mais il peut aussi créer une bibliothèque (library) destinée à réutiliser les modules objets dans d'autres logiciels.

a. Codes sources

Les codes sources C++ sont saisis à l'aide d'un éditeur de texte. Certains éditeurs de codes sources ont une fonction de coloration syntaxique, c'est-à-dire qu'ils affichent les mots clés du langage dans une certaine couleur, les chaînes littérales dans une autre couleur, les commentaires sont également distingués par une couleur spécifique, etc. L'affichage devient alors plus aisé, ainsi que la recherche d'erreurs dans le programme.

En principe, les codes sources sont saisis dans des fichiers disposant de l'extension .**cpp**. Vous pouvez également rencontrer des codes sources ayant l'extension **.h**. Il s'agit de fichiers d'en-têtes (header) destinés à être inclus par des fichiers **.cpp** à l'aide de la directive **#include**. Le préprocesseur réalise cette "insertion". L'assemblage des fichiers **.h** dans des fichiers **.cpp** ne modifie pas ces derniers. Un fichier temporaire est créé pour la durée de la compilation.

Il existe deux origines de fichiers d'en-têtes ; les en-têtes système et les en-têtes propres au programme.

Les en-têtes système contiennent des déclarations de fonctions utilitaires, telles que l'affichage de données à l'écran ou encore la manipulation de fichiers. L'implémentation de ces fichiers n'est pas fournie sous la forme de code source, mais le programmeur n'en a pas non plus besoin. Il se contentera de lier ses modules objets avec la librairie contenant les fonctions en question sous une forme compilée.

Les en-têtes propres au programme définissent des constantes, des fonctions et des classes qui sont implémentées dans des fichiers **.cpp**.

La directive **#include** s'emploie avec deux syntaxes, suivant que le fichier à inclure se trouve dans le répertoire système ou dans le répertoire du programme.

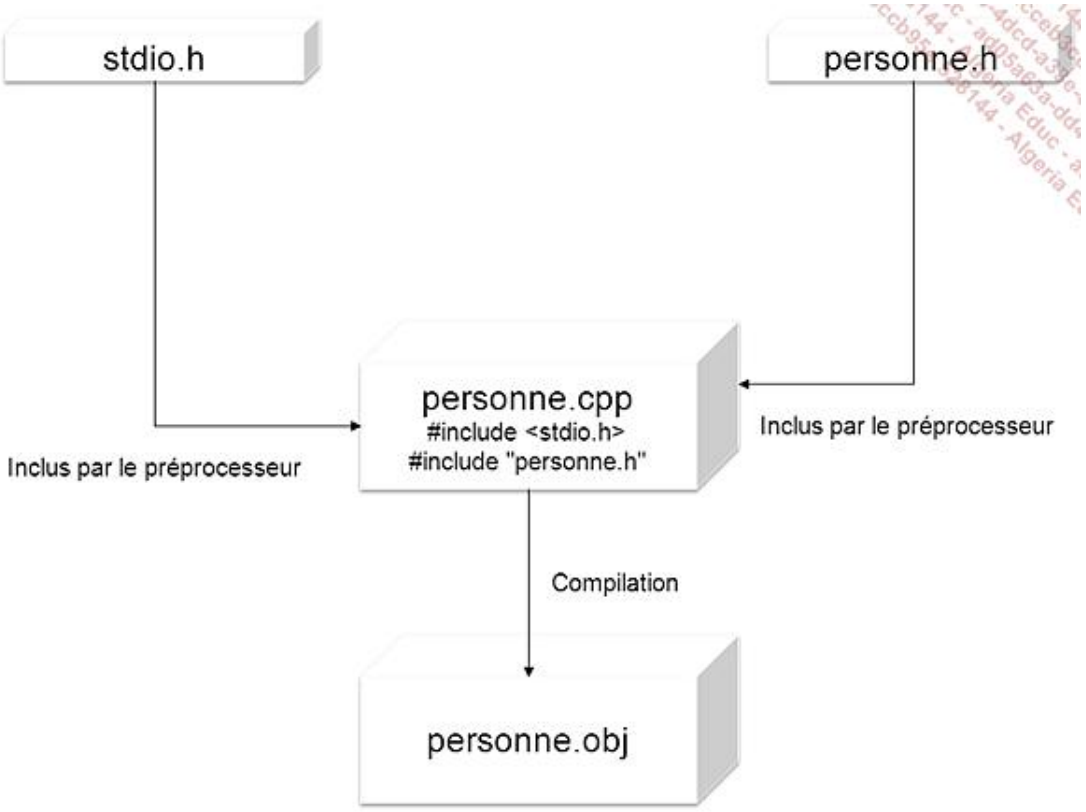
#include <stdio.h> : **stdio.h** est recherché dans le répertoire des en-têtes système

#include "personne.h" : **personne.h** est recherché dans le répertoire du programme.

Quoi qu'il en soit, les fichiers d'extension **.h** ne sont pas directement compilés. Ils sont destinés à être inclus dans un fichier **.cpp** qui sera lui compilé pour former un module objet. Le module objet porte généralement le nom du fichier **.cpp**, avec l'extension **.obj** (ou **.o** sous Unix) mais cela n'est nullement une contrainte ou une obligation.

Étudions à présent un petit exemple.

Dans cet exemple, le fichier à compiler s'appelle **personne.cpp**. Il a besoin de deux fichiers d'en-tête, **stdio.h** et **personne.h**, qui seront compilés en même temps que lui, après avoir été inclus par la directive **#include**.



Il résulte de la compilation un unique module objet appelé **personne.obj**.

Pour obtenir ce résultat, le fichier **personne.cpp** pourrait ressembler à cela :

```
#include <stdio.h>
#include "personne.h"

int main()
{

}
```

Peu importe ce que contiennent effectivement les fichiers **stdio.h** et **personne.h**.

La commande de compilation sera la suivante (avec le compilateur gcc) :

```
gcc -c personne.cpp -o personne.obj
```

Autrement dit, nous demandons au compilateur C++ **gcc** de compiler le fichier **personne.cpp** pour former le module objet **personne.obj**. Le commutateur **-c** demande à **gcc** de ne pas appeler automatiquement l'éditeur des liens, donc il se contentera de compiler **personne.cpp**.

Il est à noter que le fichier **personne.obj** n'est créé que si la compilation se passe bien. Toute erreur de syntaxe, toute absence de fichier **stdio.h** ou **personne.h** provoquera l'arrêt prématuré du processus. Le compilateur génère alors un message qui décrit les circonstances du problème, éventuellement précédé d'un numéro de ligne et du nom du fichier analysé où l'erreur s'est produite.

Par exemple, nous pourrions avoir le message suivant :

```
personne.cpp, ligne 2 : le fichier personne.h est
introuvable.
```

Parfois l'erreur a lieu dans un fichier d'en-tête qui contient des instructions ne respectant pas la syntaxe C++ :

```
personne.h, ligne 38 : erreur de syntaxe.
```

Il devient alors intéressant d'utiliser un environnement de développement intégré. Les messages d'erreurs sont affichés dans une fenêtre spéciale. Il suffit de sélectionner - généralement par double clic - un message d'erreur, et l'éditeur de texte ouvre le fichier correspondant, puis il affiche la ligne où se situe le problème.

b. Modules objets

Les modules objets sont une forme intermédiaire entre le code source C++ et l'exécutable. Le découpage en modules objets facilite la compilation séparée. Lorsque vous modifiez une ligne de programme, cette modification n'a peut-être pas d'impact sur l'ensemble du logiciel. En recompilant uniquement la partie de code "autour" de la modification, on gagne un temps considérable.

Le terme module objet n'a rien à voir avec la programmation orientée objet. Des langages tels que le langage Assembleur, le langage C, le Pascal, le Basic produisent également des modules objets. Il s'agit d'un fichier binaire résultant de la compilation d'un code source.

La question revient à choisir une stratégie pour découper au mieux son programme. Très souvent, les classes - que nous étudierons en détail dans cet ouvrage - sont définies dans un fichier d'en-tête **.h** et implémentées dans un fichier source **.cpp**. Chaque fichier source est compilé et donne lieu à un module objet.

Il est également possible de créer des modules objets à partir de fichiers sources **.cpp** regroupant un certain nombre de fonctions du logiciel. Ainsi, il n'est pas rare de trouver un module objet formé à partir du module **main.cpp**, dans lequel on a défini la fonction principale du programme **main()**.

Le langage C++ nous autorise à dépasser cette construction. Longtemps, les langages de programmation, comme le C, se sont cantonnés à cette construction où module fonctionnel équivaut à fichier source. C++ élargit cette vision :

Pour le concepteur, il importe plus d'avoir une vision logique des fonctions qu'un point de vue physique. Les espaces de noms ont été créés dans ce but. Or en C++, une classe engendre son propre espace de noms, il devenait logique de ranger chaque classe dans son propre fichier de code source. Il s'agit pourtant d'une convention et non d'une contrainte, aussi pouvez-vous définir autant de classes que vous le souhaitez dans un même fichier de code source. Il est enfin possible d'implémenter une classe définie dans un fichier **.h** au travers de plusieurs fichiers **cpp**. Toutefois, ces constructions ne sont pas très naturelles, elles représentent un risque de complication lors de l'édition des liens.

Nous allons étudier à présent la structure d'un module objet. S'il est vrai que de nombreux langages utilisent un processus de construction identique à C++ - compilation, édition des liens, exécution - les modules objets sont loin d'être compatibles entre eux. C'est parfois également vrai d'un compilateur C++ à l'autre.

Les modules objets contiennent le code source compilé. Le travail d'un compilateur consiste d'une part à effectuer des vérifications, d'autre part à générer du code (binaire, assembleur) en fonction des instructions. Le module objet contient :

- la déclaration des variables au niveau assembleur (nom, taille),
- la déclaration des fonctions (nom, code binaire).

L'éditeur des liens est chargé d'assembler (de réunir) différents modules objets pour construire une librairie (**.lib**, **.dll** ou .**so**) ou bien un exécutable (**.exe** sous Windows, pas d'extension sous Unix). L'édition des liens consiste à organiser l'assemblage en respectant le format imposé par le système d'exploitation, du moins lorsque la cible est un exécutable.

En général, toutes les variables sont regroupées dans un segment de données, et toutes les fonctions (toutes les instructions) sont installées dans un segment de code. Un en-tête complète l'ensemble en regroupant différentes informations d'ordre général (métadonnées, point d'entrée). L'éditeur des liens va ensuite remplacer chaque référence à une variable ou à une fonction par son adresse dans l'assemblage réalisé. Les

noms symboliques disparaissent au profit d'une adresse entièrement numérique.

Cette construction d'un assemblage appelle certains commentaires. En premier lieu, est-il envisageable de mélanger un module objet obtenu à partir d'un code source écrit en langage C avec un module objet obtenu à partir de C++ ? S'il est vrai que les deux langages sont proches, notamment en ce qui concerne les types de données et leur représentation, les modules objets utilisent des conventions différentes. Les noms symboliques sont formés distinctement. Par ailleurs, lors de l'appel d'une fonction, les paramètres ne sont pas transmis dans le même ordre ; enfin, les techniques de restauration de pile sont différentes. Pour résoudre ces difficultés, le programmeur peut employer des directives propres au compilateur C++ dans le but de préciser la façon d'appeler une fonction située dans un module compilé en C. Ces directives sont **`extern "C"`** et parfois aussi PASCAL. Nous le retrouverons un peu plus tard en détail.

Pour lier un module objet C++ à un module objet écrit dans un langage tel que Basic, l'affaire se corse. La situation est pourtant plus fréquente qu'on ne l'imagine. Nombre d'applications sont écrites pour partie en Visual Basic, l'interface graphique ; et pour autre partie en C++, les traitements. La première chose à considérer est alors la représentation des types de données qui varie singulièrement d'un langage à l'autre. Deux techniques sont alors envisageables : utiliser un compilateur et un éditeur de liens qui reconnaissent le format avec lequel on souhaite travailler, ou bien créer des modules objets communs à plusieurs langages de programmation.

Microsoft a longtemps favorisé la première technique mais s'est retrouvé dans une impasse qui l'a conduit à la seconde. Les compilateurs Visual Basic arrivaient à exploiter des modules C++ (composants COM ou Active X), toutefois, la vie du développeur devenait incroyablement compliquée lorsqu'il s'agissait d'échanger des données sous la forme de chaînes de caractères, de dates... Les ingénieurs de Microsoft ont ensuite conçu la plate-forme .Net, où les modules objets sont identiques d'un langage de programmation à l'autre. Ils ne se sont pas arrêtés là puisqu'ils en ont profité pour uniformiser les types de données entre leurs différents langages en introduisant la norme CTS, Common Type Specification. De ce fait, les langages VB.NET, C# et C++ partagent un grand nombre de caractéristiques communes.

Sous Unix, la situation se présente mieux car les langages C et C++ représentent la majorité des développements. Les autres langages, tels TCL/TK ou PHP sont en fait des surcouches du C (comme C++ à ses débuts), si bien que la question de l'interopérabilité entre langages se pose tout simplement moins souvent.

Reste qu'entre compilateurs C++, les modules objets n'ont pas toujours la même organisation, même lorsqu'ils sont compilés sur la même plate-forme. Nous retrouverons des exemples de situations problématiques en envisageant l'alignement des mots (unité de donnée) au niveau de l'octet, du double octet ou du quadruple.

c. Librairies (bibliothèques)

L'assemblage formé par l'éditeur de liens n'est pas toujours un exécutable. On peut livrer certains modules objets sous la forme de librairies statiques ou dynamiques.

Une librairie statique se contente de regrouper un certain nombre de modules objets. Sous Windows, on lui donne souvent l'extension **`.lib`**. Des programmes exécutables (contenant donc une fonction **`main`**) utiliseront certaines fonctions de la librairie, après ligature. Les fonctions système telles que l'affichage, la gestion des chaînes de caractères... sont livrées sous forme de librairies statiques. Certains compilateurs proposent plusieurs versions, exécution simple thread ou multithread, version internationalement neutre, version spécifique au français, en fonction des plates-formes et des compilateurs.

On emploie aussi des librairies statiques tierces d'utilité applicative, telles que les librairies d'analyse lexicale et grammaticale et les librairies mathématiques.

Au temps où la mémoire était limitée, il fallait minimiser la taille des exécutables. Pourtant les logiciels devenaient fonctionnellement de plus en plus riches. Pour résoudre cette contradiction d'objectifs, les librairies dynamiques ont été inventées. Elles permettent de mutualiser (de partager) du code et parfois aussi des variables entre plusieurs processus.

Un processus est un programme en cours d'exécution. Lorsqu'un processus appelle une librairie dynamique (**`.dll`** sous Windows et **`.so`** sous Unix), l'édition des liens doit être opérée. Il ne s'agit pourtant pas des mêmes techniques que celles employées avec un module objet ou une librairie statique. C'est en général le système d'exploitation lui-même qui assure ce rôle.

Cette opération soulève quelques remarques. Premièrement, l'éditeur des liens doit être prévenu que certaines fonctions ne seront disponibles qu'à l'exécution. Il ne doit pas chercher à fonctionner comme à l'accoutumée. On obtient ce résultat au prix d'une syntaxe quelque peu alourdie, car nombre de programmes C++ utiliseront à cet effet des pointeurs de fonction.

Deuxièmement, la localisation de la librairie au cours de l'exécution provoquera un ralentissement plus ou moins long : la librairie doit être chargée en mémoire, éventuellement initialisée, puis la fonction sera liée au programme appelant.

Les librairies dynamiques ont connu ces derniers temps un essor inattendu, tant sous Unix que sous Windows, au prix il est vrai, de substantielles modifications de leur fonctionnement et de leur définition.

Pour finir, ajoutons que le terme librairie est peut-être inapproprié, quoique généralement usité. En effet, le terme anglais library devrait être traduit par bibliothèque.

d. Exécutable

Un exécutable est le résultat fourni par défaut par l'éditeur de liens. Il suit une construction qui dépend du système d'exploitation. Certains systèmes, comme Windows, proposent différentes organisations d'exécutables : **`.com`**, **`.exe`**, **`.dll`**. Sous Unix, les choses paraissent moins variées, car l'attribution du droit d'exécution (**`--x`**) suffit à rendre éligible un fichier à l'exécution... sous réserve d'une constitution adaptée.

Quelle que soit la plate-forme, un exécutable est formé à partir d'un en-tête indiquant le point d'entrée du processus, c'est-à-dire l'adresse de la première instruction à exécuter. En principe, cette valeur varie peu, et les logiciels anti virus mettent à profit cette caractéristique pour détecter un virus informatique. Un exécutable contaminé contient en principe les instructions propres au virus à la fin du fichier, afin d'éviter de tout décaler et de tout réécrire. Le point d'entrée du processus est alors provisoirement dérouté à la fin du fichier, puis une instruction de saut reprend le cours normal de l'exécution. Ainsi, l'utilisateur lance son processus, sans se rendre compte immédiatement qu'un virus s'est installé. Son programme semble se lancer normalement ; ensuite, les stratégies virales divergent !

Quoi qu'il en soit, le format exécutable a peu évolué depuis sa conception il y a plus de vingt ans. C++ produisant un code assembleur difficile à décompiler, l'exécutable est vulnérable face aux intrusions virales, nous venons de le voir, mais aussi face aux dégradations lors de transmissions à travers les réseaux informatiques. Des éditeurs tels que Microsoft ont revu en profondeur la constitution d'un exécutable, appelé assemblage, avec la technologie C++ managé (C++ pour .NET). Dans cet environnement, l'exécutable peut être signé numériquement pour garantir à la fois son intégrité et son origine.

L'autre faiblesse des exécutables C++ réside précisément dans le manque d'information sur l'impact du fonctionnement. C++ permet certes d'écrire des systèmes d'exploitation et des compilateurs, mais il est aussi fort apprécié pour ses performances et son ouverture sur le monde applicatif. Dans un environnement d'exécution tel qu'Internet (via l'interface de programmation CGI), l'exécutable C++ constitue un risque. Impossible en effet de prédire, à la simple vue d'un fichier exécutable, s'il renferme des instructions dangereuses telles que le reformatage du disque dur. Cet écueil plaide en faveur d'une exécution sécurisée des programmes C++, comme cela se fait avec les programmes Java ou C#.

Là encore, Microsoft a devancé d'autres éditeurs en proposant un C++ managé, ce qui a été possible en aménageant quelque peu la syntaxe. On peut être surpris par la démarche, mais C++ a des qualités indéniables en termes de performances et de consolidation de l'existant, ce qui le rend de plus en plus populaire dans un contexte Internet. On voit ainsi apparaître des librairies C++ pour construire des services web. Le besoin de sécuriser l'exécutable se fait alors nettement ressentir.

5. Le préprocesseur

Le préprocesseur est chargé d'effectuer un premier traitement "textuel" sur les fichiers sources. C'est lui qui décode les lignes débutant par le symbole #. Ces lignes servent à imbriquer des fichiers de manière conditionnelle et à évaluer des constantes symboliques (également appelées macro).

En fait, le préprocesseur est devenu une partie intégrante du compilateur qui le contrôle et optimise son application.

6. Choix d'un compilateur

Le choix d'un compilateur dépend d'abord de la plate-forme envisagée. Windows ? Linux ? Macintosh ? Chacune a ses supporters.

Mais le choix n'est pas entièrement déterminé par cette question. Les librairies proposées (framework), l'environnement de développement sont également importants à prendre en compte.

Enfin, certains compilateurs se montrent meilleurs que d'autres dans des domaines particuliers, tels que l'optimisation du code ou le support complet des modèles (templates).

Voici une courte liste, loin d'être exhaustive, des compilateurs C++ les plus utilisés.

GNU g++	Licence GPL	Pratiquement incontournable pour compiler le noyau de Linux. Très respectueux du standard ANSI. Compilateur assez lent et code peu optimisé.
Microsoft C++	Licence commerciale mais édition express gratuite	Code assez optimisé. Bon support des templates. Compilateur mode ANSI et mode managé .Net (langage étendu).
Compilateur C++ Embarcadero (anciennement Borland)	Licence commerciale	Compilateur et EDI multiplate-forme Windows et Unix.
Intel C++	Licence commerciale	Code assez optimisé, notamment lorsque le microprocesseur cible est du même fabricant.
Watcom C++	Licence commerciale	Très réputé dans le monde de la création des jeux, tels que DOOM.

Pour le micro-ordinateur, le choix du compilateur sera également déterminé par le prix et la disponibilité du produit. Lorsqu'il s'agit d'une station de travail ou d'un mini-ordinateur, le choix peut se restreindre au compilateur proposé par le constructeur (HP, Sun...). On peut alors se tourner vers un projet de portage du compilateur GNU sur cette plate-forme, mais parfois le travail n'est pas encore achevé.

7. L'éditeur de liens

L'éditeur de lien s'occupe d'assembler les modules objets puis de résoudre les références symboliques en formant un exécutable ou une bibliothèque de code binaire "pur". Cet utilitaire appelé en fin de chaîne dépend beaucoup de l'environnement et du système d'exploitation.

Les bases de la programmation C++

Nous allons maintenant découvrir comment C++ permet d'implémenter des algorithmes. Ce langage appartient à la famille des langages procéduraux, ce qui signifie que les instructions d'un programme sont regroupées pour former des procédures - que l'on appelle aussi fonctions.

Un programme C++ utilise d'une part des variables pour ranger des valeurs et d'autre part des instructions pour faire évoluer ces valeurs. Ce n'est pas l'aspect le plus original de C++ puisqu'il partage cette base "algorithmique" avec le langage C. De ce fait, de nombreux types de données sont communs aux deux langages et les instructions de base sont également identiques. Ceci facilite l'apprentissage du langage C++ et améliore la portabilité ascendante.

Signalons aussi que la syntaxe C++ est un peu plus souple que celle du C, notamment en ce qui concerne la déclaration des variables et des paramètres. La relecture des programmes s'en trouve naturellement améliorée.

Pour le lecteur qui découvre la programmation orientée objet avec C++ il est essentiel d'assimiler pleinement la programmation fonctionnelle, c'est-à-dire à base de fonctions. Connaître les algorithmes de base - recherches, tris - est un très bon moyen d'y parvenir. La programmation orientée objet est un sur ensemble de la programmation fonctionnelle, une façon particulière de la structurer, de l'exploiter. Mais les règles de base demeurent les mêmes.

Avant toute instruction et toute déclaration de variable, expliquons la notation des commentaires.

Les commentaires sont des annotations rédigées par celui qui programme. Ils facilitent la relecture et rappellent parfois le rôle de certaines variables ou de certains blocs d'instructions.

Le langage C++ connaît deux types de commentaires : les commentaires sur une seule ligne et ceux qui occupent plusieurs lignes.

Pour le premier type, on utilise la barre oblique redoublée. Le compilateur qui reconnaît cette séquence `//` ignore tout ce qui suit jusqu'à la fin de la ligne.

Le second type est délimité par les séquences `/*` et `*/`, ce qui autorise l'annotation sur plusieurs lignes ou bien sur une partie de ligne seulement, comme en langage C. Attention, l'utilisation de commentaires imbriqués `/* ..... /* ... */ .....*/` n'est pas toujours acceptée par le compilateur.

Les commentaires peuvent être placés dans n'importe quel fichier source C++.

1. Déclaration de variables

a. Utilité des variables

Il existe plusieurs sortes de variables. Selon l'emploi qui leur est destiné, on déterminera un nom, une portée, un type et parfois même une valeur initiale. Ce sont les algorithmes qui font ressortir la nécessité d'employer des variables.

Lorsque vous créez une variable, vous devez toujours choisir le nom le plus explicite qui soit, même s'il est un peu long. D'une part, les éditeurs de code source disposent presque tous d'une fonction de complétion, d'autre part la lisibilité d'un programme est l'objectif numéro un.

Ainsi, pour représenter les dimensions d'un meuble, préférez les noms largeur, hauteur et profondeur aux noms sibyllins l, L et P.

Pour les boucles, on préfèrera des identificateurs (noms) plus courts, comme i, j, k à condition que leur portée soit limitée. Il est hors de question de déclarer une variable i dans une portée globale, ce serait beaucoup trop dangereux pour le fonctionnement des algorithmes.

Il est également important de souligner qu'une variable voit sa valeur évoluer au cours du temps. Une variable peut donc servir à recueillir une valeur disponible à un moment donné et à mémoriser cette valeur autant de

temps que nécessaire. Ce résultat pourra soit évoluer - la valeur de variable est modifiée - soit concourir à l'élaboration d'une autre valeur en évaluant une expression où elle intervient.

Variable d'ordre général	Représente une caractéristique attachée à un objet naturel, comme une largeur, un poids ou un prix. L'unité - mètre, kilogramme ou euro - n'est pas mémorisée par la variable elle-même. Sa valeur évolue généralement peu.
Variable discrète	Sa valeur évolue régulièrement dans une plage de valeurs, de 1 à 15 par exemple. Permet de réaliser une action un certain nombre de fois.
Variable intermédiaire	Sa valeur est actualisée en fonction de certains calculs. Une somme constitue un bon exemple de ce type de variable.

Le moyen le plus simple pour faire évoluer la valeur d'une variable est l'affectation, désignée par l'opérateur **=**. La variable est placée à gauche du signe **=** et la valeur à droite. On parle de l-value (left-value) et de r-value (right-value).

```
aire = largeur * longueur ;
```

Il existe d'autres moyens que l'affectation pour intervenir sur la valeur d'une variable, moyens que l'on désigne par effets de bord. La portée est une caractéristique déterminante pour protéger une variable contre une écriture malencontreuse.

Seule la valeur d'une variable est modifiable. Son type, sa portée et même son nom sont définis une fois pour toutes lors de sa déclaration.

b. Portée des variables

Une variable perd son contenu lorsque le flot d'exécution quitte sa portée de déclaration. Lorsqu'il s'agit d'une portée globale, elle perd son contenu lorsque l'exécution du programme prend fin.

La déclaration des variables est obligatoire dans un programme C++. Il n'est pas possible d'utiliser une variable non déclarée, le compilateur soulève dans ce cas une erreur. Seuls les langages interprétés disposent de cette caractéristique, comme Basic ou PHP.

La déclaration d'une variable spécifie toutes ses caractéristiques en une seule étape.

L'endroit où s'effectue cette déclaration détermine la portée de la variable, c'est-à-dire la région dans le code où elle a du sens. Hors de sa portée, il peut être impossible d'y accéder pour lire sa valeur ou pour la modifier.

La variable peut très bien aussi ne plus exister lorsqu'elle est considérée hors de sa portée.

Portée globale	Variable dite globale. Est en principe accessible par toutes les fonctions du programme. Nombreuses occasions de modifications concurrentes, donc l'usage de cette portée est à limiter le plus possible.
Portée d'un espace de noms	Variable moins globale, car des modificateurs d'accès - **`private`**, **`public`** - peuvent être employés pour limiter son usage hors de l'espace de noms.
Portée d'une classe	Raisonnement identique à celui de l'espace de noms, hormis le fait que la variable existe autant de fois que la classe est instanciée.
Portée d'une fonction	Variable dite locale. Dévolue à un usage algorithmique. Niveau de protection assez élevé.
Portée d'un bloc	Variable très locale. Raisonnement identique à la variable recevant la

d'instructions	portée d'une fonction.

Dans le tableau ci-dessus, l'appellation portée locale signifie que la variable est déclarée à l'intérieur d'une fonction. Dans le cas de la portée d'un espace de noms, la variable est déclarée à l'intérieur d'un espace de noms, et ainsi de suite.

```
int v_globale;  // variable globale

namespace Formes
{
  double aire;  // portée d'un espace de noms
}

class Cube
{
  float arrete;  // portée d'une classe (champ)
} ;

void compter()
{
  int i;      // variable locale
}
```

Lorsque l'on fait référence à une variable, le compilateur privilégie toujours celle qui a la déclaration la plus proche. Autrement dit, si deux variables portent le même nom, l'une étant globale et l'autre étant locale à une fonction, la variable globale sera masquée par la variable locale pour les expressions situées à l'intérieur de la fonction :

```
int i;      // variable globale

void compter()
{
  int i;    // variable locale
  i=2;      // affecte la variable locale
}
```

Pour contourner ce problème, on peut employer l'opérateur de résolution de portée **::** qui explicite la portée à laquelle se rapporte la variable considérée :

```
int i;
void compter()
{
  int i;    // variable locale
  i=2;      // affecte la variable locale
  ::i=8;    // variable globale affectée
}
```

Nous trouverons par la suite d'autres emplois de cet opérateur **::** qui n'existe pas dans le langage C.

c. Syntaxe de déclaration

La syntaxe pour déclarer une variable est très concise :

```
type nom ;
```

Le type et le nom sont obligatoires.

Le point-virgule clôt la déclaration, il est nécessaire. On trouve parfois aussi une déclaration procédant à l'initialisation de la variable avec une valeur :

```
int prix = 30 ;  // déclare une variable prix de type entier
```

Il est également possible de partager le type entre plusieurs variables, dans le but de raccourcir la séquence de déclaration :

```
int largeur, hauteur ;
```

Naturellement, chaque variable largeur et hauteur est de type entier (**int**). Chacune d'entre elles peut recevoir une valeur initiale :

```
int largeur = 30, hauteur = 120 ;
```

Le nom de la variable doit commencer par une lettre ou par le caractère de soulignement. Sont exclus des identificateurs de variable les caractères assimilés à des opérateurs, tels que **- + / . <** ainsi que le caractère **$**.

Il n'est pas judicieux d'employer des caractères accentués, même si le compilateur les accepte. La portabilité du programme serait plus que réduite.

d. Types de données

Le langage C++ reprend les types primitifs du langage C. Ces types se répartissent dans différentes catégories : types entiers, types décimaux, caractères. C++ ajoute un type booléen et n'améliore pas la représentation archaïque des chaînes de caractères. Une classe est plutôt proposée dans la bibliothèque standard STL mais son emploi n'est pas aussi généralisé que l'habituel **char***.

Pour les types dont le codage varie d'un compilateur à l'autre (**int** par exemple), l'opérateur **sizeof()** retourne la largeur du type ou de la variable en octets.

Les types entiers

Le langage C a été inventé à une époque où l'informatique était très terre à terre. Par ailleurs, il a été conçu pour tirer parti des instructions spécialisées du processeur animant le PDP-11 (un ancêtre du Vax de Digital). Certaines notations comme le fameux ++ viennent directement de cette "optimisation". Côté variable, l'unité de base reste l'octet. Cette quantité est assez bien placée pour représenter des caractères ASCII (la table de base contient 128 symboles) mais aussi des entiers courts, entre -128 et +127.

Le compilateur utilise la représentation en complément à deux, ce qui lui permet de traiter nombres négatifs et soustractions avec la même arithmétique que l'addition.

Prenons le type **char**, qui autorise la représentation de 2^8 = 256 valeurs. Ces valeurs sont réparties entre -128 et +127, en complément à deux. Le complément à deux est en fait le complément à un (inversion de tous les bits) augmenté d'une unité. Avec des variables de type **char**, additionnons 4 et -3 :

```
char x = 4 ;
char y = -3 ;
char z = x + y ;
```

Pour avoir le codage de la valeur 4, c'est-à-dire de la variable x, il suffit de décomposer en base 2, nous obtenons $1x2^2$ = 4, donc :

```
x = 0000 0100
```

Pour avoir le codage de la valeur -3, c'est-à-dire de la variable y, commençons par écrire 3 en base, soit $1x2^1$ + $1x2^0$ = 3, soit :

```
+y = 0000 0011
```

Puis inversons tous les bits, nous obtenons le complément à un :

```
!+y = 1111 1100
```

Ajoutons 1, la représentation de -3, en binaire et en complément à deux, est :

```
y = 1111 1101
```

À présent, additionnons x et y :

```
x = 0000 0100
y = 1111 1101
```

L'addition bit à bit, avec propagation des retenues, donne :

```
z = 0000 0001
```

Ce qui est conforme au résultat attendu.

Le format en complément à deux a été choisi pour sa simplicité. C++ propose un certain nombre de types de données pour représenter des nombres entiers, dont la largeur varie de un à huit octets, en fonction du microprocesseur.

char	256 valeurs [-128 ; +127]	Sert aux octets et aux caractères.
short	32768 valeurs [-32 767 ; +32 768]	Entier court, abréviation de **short int**. Avec les processeurs 32 bits, son emploi tend à disparaître, sauf pour des raisons de compatibilité.
int	2^{32} valeurs [$-2^{31}-1$; $2^{31}-1$]	Entier. En principe, sa largeur adopte la taille du mot machine, donc 32 bits le plus souvent.
long	2^{64} valeurs [$-2^{63}-1$; $2^{63}-1$]	Entier long, abréviation de **long int**. En principe le double de l'**int**, mais parfois limité à 32 bits avec certains compilateurs !

L'emploi du qualificateur **unsigned** déplace les valeurs représentées dans l'intervalle [0 ; 2^n-1]. La déclaration d'une variable devient alors :

```
unsigned char c ;
```

Une des grandes difficultés de portage d'application C++ vient de la multitude des formats de représentation attachés aux types. Il est vrai qu'entre un microcontrôleur 8 bits et un Power PC, le type **int** n'aura pas la même largeur. Mais cette situation n'est pas toujours avérée et les compilateurs 16 bits finissent par disparaître.

Pour les variables discrètes - utilisées dans les boucles for - il peut être opportun d'employer le type **int**. En effet, les microprocesseurs 32 bits, les plus répandus, doivent ralentir pour traiter des quantités inférieures. Ils lisent la mémoire par blocs de quatre octets, voire davantage. Il y a toutes les chances que la variable soit en fait logée dans un registre du microprocesseur, mais leur nombre est compté, surtout avec des microprocesseurs utilisant un registre d'instructions complexe, les CISC, dont le Pentium fait partie. Par

opposition, les processeurs RISC tels que le Power PC disposent de peu d'instructions mais de beaucoup de registres. Ils sont donc avantagés par le compilateur pour traiter les boucles.

Lorsque vous spécifiez une valeur littérale de type entier, le compilateur utilise la lecture en base 10. Donc l'affectation **`x = 123`** signifie que vous placez la valeur 123 dans la variable x.

Comme C++ reprend les caractéristiques de C et comme C a été inventé de manière concomitante d'Unix, le compilateur reconnaît aussi les bases octale et hexadécimale.

En octal, chaque chiffre évolue entre 0 et 7, puisque la base est 8. Le compilateur reconnaît l'emploi de l'octal en préfixant la valeur littérale par un zéro :

x = 0123 équivaut en fait à $x = 1x8^2 + 2x8^1 + 3x8^0$ soit 83 en décimal.

Le format octal est surtout utile pour représenter les droits d'accès aux fichiers Unix. Ces droits définissent la valeur de 3 bits, **`rwx`** (**`read write execute`**), ce qui donne 8 possibilités et entraîne l'emploi de la base octale.

L'hexadécimal est plus commode pour manipuler la base 2, très risquée pour l'interprétation humaine. Dans cette base, on utilise les dix chiffres 0 à 9 auxquels on adjoint les lettres A, B, C, D, E et F pour compter jusqu'à 15. Les seize possibilités d'un digit hexadécimal font que ce chiffre est équivalent à un quartet, soit un demi-octet. Le compilateur reconnaît l'emploi de l'hexadécimal pour une valeur littérale par le préfixe 0x.

Avec a = 0xA28, la variable a reçoit la valeur $10x16^2 + 2x16^1 + 8x16^0$, soit a = 600 en décimal.

L'écriture binaire est assez aisée à déduire, puisque chaque chiffre représente un bloc de quatre bits :

```
1010 0010 1000
```

Les types décimaux

Le terme de nombre à virgule flottante est sans doute plus approprié que celui de réel choisi pour représenter ce nombre. C++ dispose des types **`float`** et **`double`**, conformes au format IEEE 754. Ce format standard est surtout employé par les coprocesseurs arithmétiques.

La précision est loin d'être bonne, surtout si la valeur absolue des nombres à représenter est élevée.

Le format **`float`** utilise quatre octets pour représenter un nombre à virgule. Le double multiplie par deux cette quantité. L'intérêt du double est certainement d'améliorer la précision des calculs, plus que d'élargir la zone de couverture.

Enfin, le format long double améliore encore la précision mais le plus souvent donne lieu à des calculs émulés par un logiciel plutôt que d'être confiés au coprocesseur arithmétique.

`float`	32 bits, 7 chiffres significatifs	$+/- 3,4x10^{38}$
`double`	64 bits, 11 chiffres significatifs	$+/- 1,7x10^{308}$
`long double`	80 bits, 15 chiffres significatifs	$+/- 1,2x10^{4932}$

Avant d'employer ces types dans vos programmes, ayez à l'esprit les indications suivantes :

Maniez des quantités ayant des valeurs absolues comparables. Les calculs se faisant avec une mantisse de 40 bits environ, l'addition d'un nombre très grand et d'un nombre très petit provoquera des erreurs de calcul.

Les nombres entiers sont toujours représentés exactement avec ces formats.

Certains nombres décimaux tels que 1,45 n'admettent pas de représentation exacte avec les formats **`float`** et

double. Des heuristiques intégrées aux coprocesseurs arithmétiques tiennent compte de ce phénomène, mais les calculs peuvent être faux.

La précision diminue rapidement lorsque la valeur absolue augmente. Utiliser un type **float** à la limite de sa représentation peut être dramatique pour la qualité des calculs. Passer à un double donnera des résultats plus justes, avec un surcroît de temps de calcul négligeable.

En résumé, ces formats sont utiles lorsque la compatibilité avec un logiciel extérieur est en jeu, ou bien lorsque la vitesse des calculs est un facteur déterminant. Pour les applications scientifiques ou financières, il peut être utile d'employer une bibliothèque spécialisée pour représenter les nombres sous forme de chaînes de caractères.

Le booléen

Le langage C ne connaît pas les booléens en tant que type, mais a détourné l'arithmétique entière pour évaluer ses prédicats.

Un booléen est une valeur qui est évaluée comme vraie ou fausse. Les instructions conditionnelles, **if**, **while**, **for**... fonctionnent avec un test qui doit être de type booléen.

La convention, lorsque les entiers sont utilisés comme booléens, est que 0 équivaut à faux et n'importe quelle autre valeur, strictement non nulle, équivaut à vrai.

Comme l'affectation produit une valeur à gauche, parfois de type entier, les concepteurs ont dû distinguer l'affectation de la comparaison stricte de deux quantités :

```
if(x=3)
```

Cette dernière clause est toujours vérifiée, car 3 est non nul. La variable x reçoit cette valeur 3, donc le test est positif (vrai). L'écriture correcte est sans aucun doute :

```
if(x==3)
```

L'opérateur **==** teste l'égalité de la variable x avec la valeur 3. Il renvoie vrai ou faux, selon le cas.

Le compilateur C++ soulève un avertissement (warning) lorsqu'il identifie cette écriture équivoque mais néanmoins légale d'un point de vue syntaxique. Ceci dit, le langage C++ dispose aussi d'un vrai type booléen pour évaluer des prédicats :

bool	**true** **false**	Les valeurs **true** et **false** sont des mots clés du langage, pas des macros comme on procédait parfois avec C.

Les caractères et les chaînes

Une chaîne de caractères est une suite de caractères. Le compilateur range les caractères les uns après les autres et termine la série par un caractère de code 0. On appelle ce format chaîne ASCIIZ, avec un Z pour zéro terminal.

Le type qui désigne les chaînes de caractères est habituellement **char***, mais les tableaux **char[ ]** peuvent aussi convenir sous certaines conditions.

Lorsque l'on désigne une chaîne littérale, délimitée par les caractères guillemets, le compilateur alloue une adresse pour stocker les caractères de la chaîne. Cette adresse est en fait une adresse de caractère(s), d'où le type **char***. Il est naturellement possible de lire cette zone mémoire, mais la modification des caractères contenus dans cette zone n'est pas toujours possible, de même que l'écriture au-delà du zéro terminal.

Il est possible de créer un tableau de caractères (**char[ ]**) assez vaste ou d'allouer une zone mémoire pour

recopier la chaîne littérale et pour se livrer à toutes les modifications nécessaires.

Le compilateur C++ reconnaît comme son aîné le C les séquences d'échappement.

\n	Nouvelle ligne	**\b**	Sonne la cloche (bell)
\r	Retour chariot	**\t**	Tabulation

Les valeurs littérales de caractère sont délimitées par le signe ' comme l'indique l'extrait suivant :

```
// trois caractères
char a = 65;
char b = 'B';  // soit le code ASCII 66
char tab = '\t';

// deux chaînes
char* s = "bonjour";
char* u = "comment \n allez-vous ?";
```

Pour la chaîne **u**, la séquence d'échappement a été "décollée" pour améliorer la lisibilité. Il est tout à fait possible de supprimer les espaces de part et d'autre.

Pour copier des chaînes, concaténer d'autres chaînes, calculer la longueur, effectuer des recherches, la librairie **<string.h>** fournit toutes les fonctions nécessaires.

Le langage C++ offre aussi une prise en charge des chaînes plus évoluée par le biais de sa librairie standard, la STL.

2. Instructions de tests

Indispensables à l'énoncé d'un programme, les opérateurs et les instructions sont les éléments qui permettent de traduire un algorithme en C++.

Les opérateurs combinent différentes valeurs pour aboutir à des expressions. Ces expressions sont de type numérique (entier, décimal) ou booléen. Lorsque les opérateurs sont surchargés, d'autres types peuvent découler de l'application d'opérateurs.

a. Instructions de tests

C++ connaît deux types de tests : les tests simples et les tests multiples. Le test simple est un basique de l'algorithme, il s'agit de l'instruction **if**.

Le test multiple a été conçu afin de tirer parti de l'instruction **switch** du PDP-11. Il s'agit d'une optimisation, évitant de tester plusieurs fois une même valeur entière. De nos jours, l'instruction **switch** serait plutôt un confort de programmation, mais nombre de langages dont C++ fait partie s'en tiennent à l'approche du langage C.

Le test simple, if

Cette instruction exécute conditionnellement une autre instruction, après évaluation d'un prédicat booléen. Le style de rédaction est important pour la lisibilité du programme, mais n'influe pas sur son fonctionnement.

```
if( prédicat )
    instruction
```

Le décalage d'une tabulation indique au lecteur que l'instruction n'est exécutée qu'à condition que le prédicat

soit vérifié (vrai). Une instruction peut représenter plusieurs choses :

```
instruction -> ;
    instruction_simple
    { instruction instruction_simple }
```

Autrement dit, "instruction" peut désigner soit l'instruction vide (**;**), soit une instruction élémentaire comme une affectation ou un appel de fonction, ou bien être un bloc d'instructions délimité par les accolades **{** et **}**.

Il est à noter la présence de parenthèses pour délimiter le prédicat, ce qui a permis d'éliminer le mot clé **then** prévu par les langages Basic et Pascal.

Si l'instruction se résume à une instruction simple, il vaut mieux éviter les accolades pour ne pas surcharger le programme.

```
if(x==3)
  printf("x vaut 3") ;
```

Le test simple avec alternative if ... else

Il est parfois nécessaire de prévoir l'alternative, c'est-à-dire l'instruction exécutée lorsque le prédicat n'est pas vérifié (faux).

C'est précisément la vocation de la construction **if ... else ...**.

```
if(prédicat)
  instruction_vrai
else
  instruction_faux
```

Bien sûr, les règles applicables à la définition d'une instruction simple ou composée demeurent vraies : n'employer les accolades que si cela s'avère nécessaire.

```
if(rendez_vous<mercredi)
  printf("rendez-vous en début de semaine");
else
{
  printf("rendez-vous en fin de semaine\n");
  printf("éviter le week-end");
}
```

Le test multiple switch

Le test multiple prévoit les différentes valeurs que peut prendre une expression (ou une variable). Normalement, ces valeurs sont de type entier et sont connues au moment de la compilation.

L'instruction fonctionne en énumérant les différentes valeurs possibles, appelées cas. Une clause default s'applique lorsque la valeur testée n'a satisfait aucun des cas prévus.

```
switch(valeur_entière)
{
  case valeur1:
    instruction
  case valeur2:
    instruction
  ...
  default:
    instruction
}
```

En principe, l'instruction doit se terminer par un break, autrement, l'exécution continue jusqu'à rencontrer cet arrêt, quitte à franchir de nouvelles lignes case. Cette construction particulière permet de grouper plusieurs cas :

```
switch(x)
  {
  case 1:
  case 2:
    printf("x vaut 1 ou 2");
    break;
  case 3:
    printf("x vaut 3");
    break;
  case 10:
    printf("x vaut 10");
    break;
  default:
    printf("x est différent de 1,2 et 10");
    break;
  }
```

La clause default n'a pas besoin de figurer en dernière position, mais c'est souvent là qu'on la trouve. Aussi, l'instruction **`switch`** n'est pas obligée d'inclure une clause default.

b. Opérateurs

Les opérateurs combinent différentes valeurs. Suivant le nombre de valeurs réunies par l'opérateur, on a affaire à des opérateurs unaires ou binaires. Ces valeurs prennent alors le nom d'opérandes.

Cette section dresse une liste des principaux opérateurs. Les opérateurs spécifiques aux classes, aux tableaux et aux fonctions seront abordés ultérieurement.

Opérateurs de comparaison

Ces opérateurs vérifient une relation d'ordre entre deux opérandes. Les opérandes peuvent être de n'importe quel type, mais, par souci d'homogénéité, il vaut mieux que les types soient appariés.

L'évaluation de l'expression faisant intervenir un opérateur de comparaison produit un résultat de type booléen, vrai ou faux.

==	égalité des valeurs (ne pas employer l'opérateur d'affectation =)
!=	différence des valeurs (équivaut au signe mathématique $\neq$)
<	inférieur à
>	supérieur à
<=	inférieur ou égal
>=	supérieur ou égal

Pour améliorer la lisibilité, il est recommandé d'utiliser des parenthèses pour éclaircir un prédicat faisant intervenir plusieurs opérateurs.

Opérateurs arithmétiques

Il s'agit bien entendu des quatre opérations de base - addition, soustraction, multiplication, division -

applicables à tout type numérique, entier ou non. Les entiers connaissent en plus le reste de la division entière, que l'on appelle parfois modulo.

++	addition
-	soustraction
*****	multiplication
/	division
%	modulo

Là encore, ne pas hésiter à ajouter des parenthèses pour améliorer la lisibilité, même si certains opérateurs ont une préséance plus importante que d'autres (* sur +, par exemple).

Opérateurs logiques

Il faut faire attention à ne pas les confondre avec les opérateurs booléens. Les opérateurs logiques travaillent au niveau du bit.

~	Complément à 1 (non logique)
|	ou logique
&	et logique
^^	ou exclusif
<<	décalage à gauche, n bits
>>	décalage à droite, n bits

À chaque fonction logique correspond une table de vérité, nous commençons avec le "non logique" :

b	**~b**
0	1
1	0

Découvrons à présent le "ou logique" :

b1	**b2**	**b1 | b2**
0	0	0
0	1	1
1	0	1
1	1	1

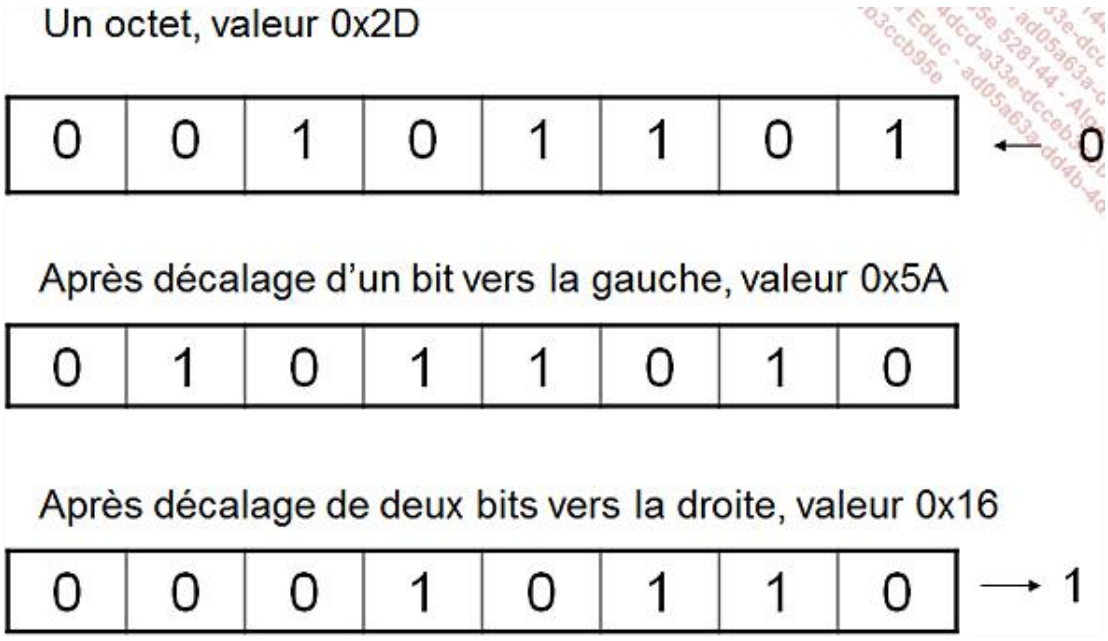
Voici maintenant le "et logique" :

b1	b2	b1 & b2
0	0	0
0	1	0
1	0	0
1	1	1

Puis finalement le "ou exclusif"

b1	b2	b1 ^ b2
0	0	0
0	1	1
1	0	1
1	1	0

Pour comprendre les opérateurs de décalage, un schéma sera plus explicatif :



Le programme correspondant ressemble à ce qui suit :

```
int main(int argc, _TCHAR* argv[])
{
  char c = 0x2D;
  printf("%x\n",c);    // 2D

  char c1 = c << 1;
  printf("%x\n",c1);    // 5A

  char c2 = c1 >> 2;
  printf("%x\n",c2);    // 16
  return 0;
}
```

Il est à remarquer que le décalage vers la gauche multiple la quantité par 2, alors que le décalage à droite divise la quantité par 2. Si le décalage a lieu sur plusieurs bits, la quantité est modifiée dans un facteur de 2 puissance le nombre de bits décalés.

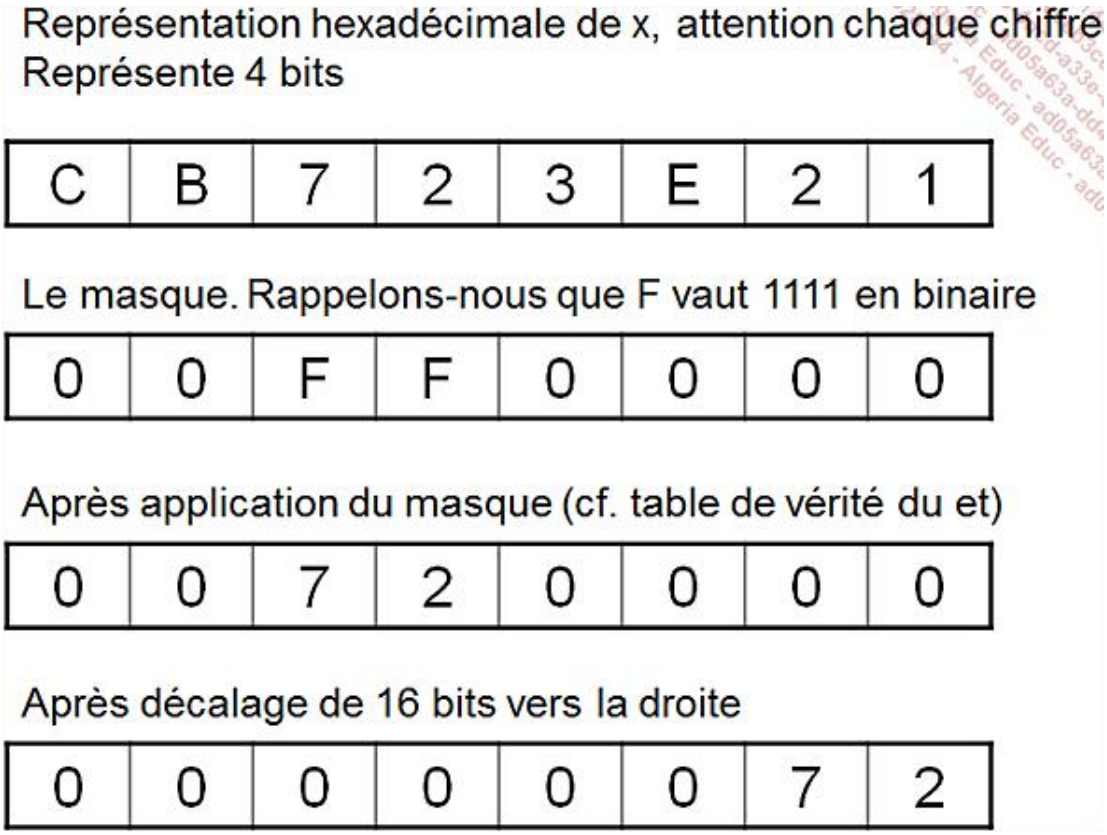
Vérifions ceci dans l'exemple donné : 5A = 5x16 + 10 = 90 et 2D = 2x16 + 13 = 45. Et aussi, 16 = 1x16 + 6 = 22, ce qui est bien le quart de 90, en arithmétique entière.

Les opérateurs logiques sont particulièrement utiles lorsque l'on structure un entier (ou un octet), ou bien lorsque l'on traite un flux binaire. Les bibliothèques de connexion au réseau en ont également besoin.

Ils sont beaucoup utilisés pour réaliser des masques de bits, afin de lire telle ou telle partie d'une valeur entière :

```
int x = 0xCB723E21;
c = (x & 0x00FF0000) >> 16;
printf("%x\n", c);  // affiche 72
```

Pour comprendre l'exécution de ce programme, suivons les calculs sur un dessin :



Évidemment, les 24 bits de poids fort sont perdus lors de l'affectation de la variable c, de type char, donc large de 8 bits. Mais on sait aussi que ces 24 bits sont nuls.

Opérateurs booléens

Les opérateurs booléens ressemblent aux opérateurs logiques si ce n'est qu'ils fonctionnent sur des valeurs de booléens et non sur des bits. Bien entendu, ils reprennent les mêmes tables de vérité mais le concepteur du langage C a dû introduire des notations spéciales pour distinguer les deux domaines. On se souvient en effet que le langage C assimile les entiers (formés de bits) et les booléens. Le langage C++ a conservé cette approche pour des raisons de compatibilité.

!	négation booléenne
&&	et booléen
||	ou booléen

Il faut remarquer la disparition du ou exclusif, que l'on peut construire à partir des autres opérateurs :

```
Xor(p,q) = (p || q) && ! ( p && q)
```

Notez également le changement de notation pour la négation. Le tilde, au lieu d'être doublé, se transforme en point d'exclamation.

Pour mettre en œuvre ces opérateurs, on peut soit faire intervenir un booléen, soit employer le résultat de l'évaluation d'un opérateur de comparaison :

```
bool p,q;
int x, y;
...
p = q && (x<y) ;
```

Comme toujours, les parenthèses sont appréciables pour améliorer la lisibilité. On peut aussi travailler sur la mise en page pour faciliter la mise au point d'un prédicat un peu long :

```
if((x < 3 ) && f(34,2*p)<fgetc(fi) || ! (k %2==0))
          ...
```

devrait s'écrire :

```
if((x < 3 ) &&
   f(34,2*p)<fgetc(fi) ||
   ! (k %2==0))
```

Les opérateurs d'incrémentation

Ces opérateurs nous viennent directement de l'assembleur du microprocesseur équipant le PDP-11. En tant que tels, il s'agit juste d'une notation commode pour augmenter ou diminuer d'une unité une variable. Mais ils peuvent devenir très efficaces avec certains processeurs CISC, pour peu que le mode d'adressage "relatif indexé" soit disponible.

Pour l'heure, nous nous concentrons sur la syntaxe :

var++	Évalue la valeur de la variable puis augmente la variable d'une unité.
var--	Évalue la valeur de la variable puis diminue la variable d'une unité.
--var	Diminue la valeur de la variable puis évalue sa nouvelle valeur.
++var	Augmente la valeur de la variable puis évalue sa nouvelle valeur.

Dans un certain nombre de situations, les notations préfixées et postfixées donneront le même résultat :

```
int x = 3;
++x;    // ici x++ est tout à fait équivalent
printf("x=%d\n",x);  // affiche x=3
```

Dans d'autres cas, l'ordre est très important :

```
int x=1,y=2;
y=(++x);  // x=2, y=2
y=(x++);  // x=3, y=2
printf("x=%d, y=%d",x,y);  // x=3, y=2
```

3. Instructions de boucle

Les instructions de boucle définissent des instructions qui sont exécutées plusieurs fois. Le nombre d'exécutions - que l'on appelle itérations - est prévu par une condition.

Pour choisir le type de boucle approprié, il faut toujours se laisser guider par la nécessité algorithmique. Si l'on connaît à l'avance le nombre d'itérations, c'est une boucle **for**. Dans les autres cas, c'est une boucle **while** (ou **do**).

a. La boucle for

Cette instruction est un peu une curiosité. Elle implémente la structure de contrôle de flot d'exécution *pour* définie par l'algorithmie impérative, mais il s'agit en fait d'une instruction **while** déguisée.

```
for(initialisation ; prédicat ; incrément)
  instruction
```

L'initialisation sert généralement à affecter des variables. Le prédicat est le plus souvent basé sur un opérateur de comparaison tel que **<**. L'incrément sert à augmenter une variable discrète.

Prenons l'exemple d'une boucle **for** prévue pour exécuter un certain nombre d'itérations :

```
for(int i=1; i<=10; i++)
    printf("i=%d\n",i);
```

Nous aurions pu écrire cela à l'aide d'une boucle **while** :

```
int i=1;
while(i<=10)
{
  printf("i=%d\n",i);
  i++;
}
```

Toutefois, l'écriture **for** est plus concise.

Les programmeurs C et C++ ont l'habitude de faire commencer i à 0, ce qui change l'expression du test :

```
for(i=0; i<10; i++)
    ...
```

Si l'on compte bien, il y a toujours 10 itérations. Nous verrons par la suite que les tableaux sont indexés à partir de 0 et que cette habitude arrange bien les programmes. Par contre, si vous traduisez d'un programme Pascal ou Basic en C++, il faut bien surveiller ces valeurs limites.

Dans la boucle **for**, toutes les parties sont optionnelles, mais les points-virgules sont toujours là :

```
for( ;  ; )
    ...
```

D'autres instructions, telles que **break**, aident alors à sortir de la boucle.

b. La boucle while

L'instruction **while** est à privilégier lorsque l'on ne connaît pas à l'avance le nombre d'itérations.

```
while( prédicat )
  instruction
```

Bien que **while** fonctionne parfaitement avec une variable discrète, il n'est pas rare de trouver des exemples qui ressemblent à cela :

```
while( L != null )
{
  printf("%s",L->element);
  L=L->suite;
}
```

Il est à remarquer que l'instruction associée au **while** peut très bien ne pas être exécutée, si le prédicat se révèle faux d'emblée.

c. La boucle do

Au contraire de la boucle **while**, le **do** provoque au moins l'exécution d'une itération, car l'évaluation du prédicat se fait au terme de cette exécution :

```
do
  instruction
while ( prédicat ) ;
```

Un exemple d'utilisation de cette boucle consiste à lire des entrées utilisateur :

```
char c;
do
{
  printf("Confirmer O/N");
  scanf("%d",&c);
} while(c!='o' && c!='O' && c!='n' && c!='N');
```

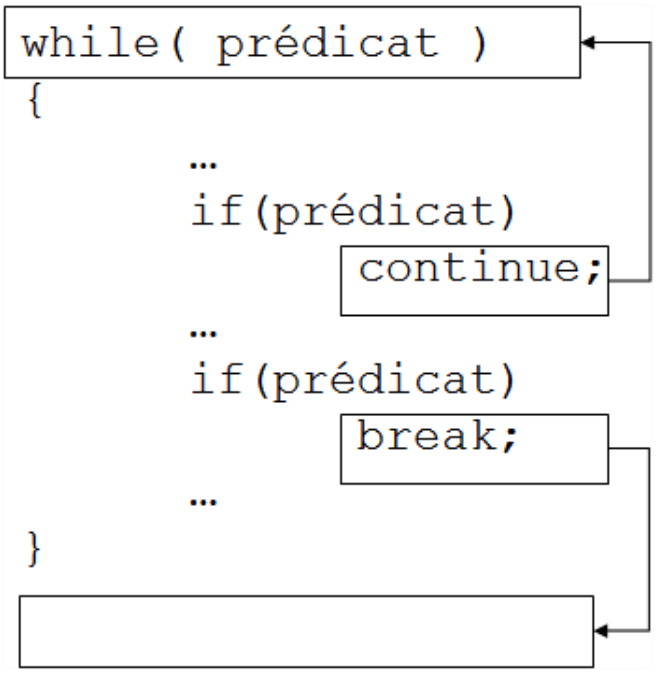
d. Les instructions de débranchement

Le langage C++ compte trois instructions de débranchement impératif : **goto**, **continue** et **break**.

Seules **break** et **continue** sont toujours usitées. L'instruction **goto** a depuis longtemps fait la preuve des complications qu'elle entraîne pour la maintenance et la relecture des programmes.

L'instruction **break** sert à sortir d'une structure de contrôle, **switch** ou boucle. L'instruction **continue** ne s'applique qu'aux boucles, elle démarre une nouvelle itération.

Il est toujours possible de construire ses programmes pour éviter l'emploi de **continue** et de **goto**. L'instruction **break** par contre améliore la lisibilité du programme.



Il n'est pas rare de combiner ces instructions avec des tests (**if**), mais cela n'est pas une nécessité imposée par la syntaxe.

4. Tableaux

Les tableaux constituent, tout comme les variables, une structure très importante pour les algorithmes. Un tableau est un ensemble de valeurs d'un type déterminé. On accède à chaque valeur en communiquant au tableau un numéro que l'on appelle index. En C++, les index débutent à 0 et croissent jusqu'à N-1, où N est la taille du tableau, c'est-à-dire le nombre d'éléments qu'il contient.

Le type du tableau - en fait le type des éléments qu'il contient - est quelconque : valeur scalaire, objet, pointeur... Voici deux exemples de tableaux :

```
double coord[2];  // un tableau de deux double coord[0]
et coord[1]
char* dico[10000]; /* un tableau de 10000 pointeurs dico[0] à dico[9999] */
```

Il est également possible de définir des tableaux multidimensionnels :

```
double matrice[3][3];  // une matrice, 2 dimensions
```

Pour initialiser un tableau, on peut accéder à chacun de ses éléments :

```
coord[0]=10;
coord[1]=15;
```

Il est également possible d'initialiser le tableau en extension, c'est-à-dire en fournissant ses valeurs au moment de la déclaration :

```
char separateurs[] = { ' ','*','-' };
```

Pour l'initialisation des tableaux de char, les littérales de chaînes rendent bien des services. L'écriture suivante :

```
char chaine[]="Bonjour";
```

est plus commode à employer que :

```
char chaine[]={'B','o','n','j','o','u','r'};
```

Voici maintenant l'exemple d'une recherche de la plus petite valeur dans un tableau de type double :

```
double m;  // valeur mini
double tab[] = {-3, 8.2, 5, 57, -11, 1.4 };

int i;
m=tab[0];

for(i=0; i<6; i++)  // 6 valeurs dans le tableau
  if(tab[i]<m)
    m=tab[i];

printf("La plus petite valeur est %f",m);
```

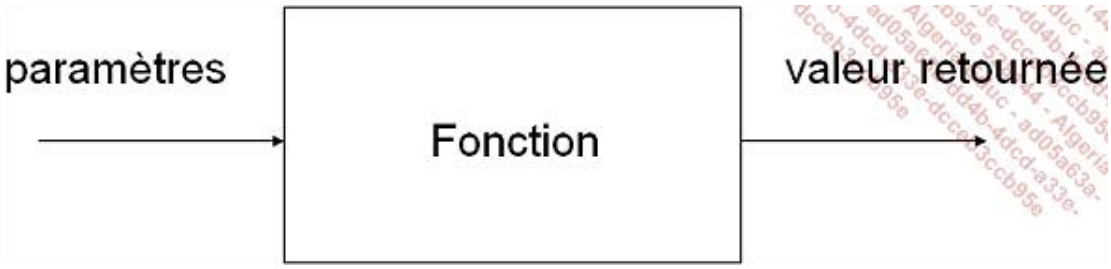
La syntaxe de déclaration d'un tableau repose sur l'emploi des crochets **[** et **]**. Nous verrons un peu plus loin que les pointeurs et les tableaux dégagent une certaine similitude.

Les tableaux déclarés avec la syntaxe **[]** sont alloués dans la portée courante ; s'il s'agit d'une variable locale à une fonction ou d'une méthode, ils sont alloués sur la pile. À l'extérieur ils peuvent être alloués au niveau du tas (heap) global ou du segment de l'objet qui les porte.

5. Fonctions et prototypes

Pour appréhender convenablement la programmation orientée objet, il est nécessaire de bien maîtriser la programmation fonctionnelle. Une fonction est un ensemble d'instructions - et parfois aussi de variables locales - auquel on a donné un nom. Cette fonction admet des paramètres et généralement évalue une valeur en retour.

Une fonction est donc modélisée par une boîte avec des entrées et une sortie :



La signature de la fonction définit son nom, ses paramètres d'entrée et sa sortie :

```
int somme(int a,int b)
```

Cette séquence précise que la fonction **somme()** reçoit deux paramètres de type entier **a** et **b** (entrée) et qu'elle retourne un entier. Peu importe pour l'instant comment s'exécute la fonction, c'est-à-dire comment la somme est déterminée.

a. Déclaration d'une fonction

En C++, il faut déclarer une fonction avant de l'utiliser. Deux moyens sont prévus à cet effet : la définition complète et le prototype. La définition complète débute par la signature et se poursuit par la liste des instructions entre deux accolades :

```
int somme(int a,int b)
{
  return a+b;
}
```

Dans l'approche par prototype, on termine la déclaration par un point-virgule.

```
int somme(int a,int b) ;
```

La signature et le corps de la fonction (donc la définition complète) peuvent ainsi figurer dans un autre fichier source d'extension **.cpp**. Le compilateur n'a pas besoin d'eux pour compiler, car le prototype le renseigne déjà sur la liste des paramètres et le type de retour. Au moment de l'édition des liens, "tous les morceaux sont recollés".

Il n'est pas rare de regrouper tous les prototypes dans un fichier d'en-tête **.h**, et de fournir l'implémentation dans un fichier **.cpp**, ou bien sous forme d'une librairie statique **.lib**.

b. Fonctions et procédures

La même syntaxe est utilisée pour décrire ces deux éléments, alors que des langages tels que Basic (**Function/Sub**) ou Pascal (**Function/Procedure**) les distinguent. En C++, comme en C, on emploie le type de retour **void**, terme emprunté à la langue anglaise et qui signifie vide, vacant. Ce type indique qu'une fonction ne renvoie rien (il s'agit d'une procédure) ou bien que l'on n'a pas encore d'information sur le type finalement employé (cas des pointeurs **void***).

Le corps d'une fonction contient un certain nombre d'instructions **return** qui stoppent son exécution et retournent une valeur d'un type conforme à celui de la fonction :

Type de fonction	Type de retour	Exemple
`void f(...)`	`-`	`return;`
`int f(...)`	`int`	`return 2;`
`bool f(...)`	`bool`	`return (x==2);`
`char* f(...)`	`char*`	`return "bonjour";`

Pour une procédure, il est possible d'omettre l'instruction **return**. L'exécution prend fin après la dernière instruction, celle qui précède l'accolade fermante. Pour une fonction, il est nécessaire de renvoyer une valeur conforme au type annoncé.

Par ailleurs, une fonction (ou une procédure) peut contenir plusieurs instructions **return**, ainsi que l'explicite l'exemple qui suit :

```
enum age { enfant, adolescent, adulte };

age categoriser(int age)
{
  if(age<13)
    return enfant;  // on s'en va

  // continue sur age>=13
  if(age<18)
    return adolescent; // on s'en va

  // continue sur age>=18
  return adulte;    // on s'en va
}
```

```
int main(int argc, char* argv[])
{
  int old;
  scanf("%d",&old);  // lire un entier depuis le clavier

  age a;
  a=categoriser(old); // déterminer la tranche d'âge

  printf("age=%d",a);  // afficher les résultats
  return 0;
}
```

c. Appel des fonctions

Pour appeler une fonction depuis une autre fonction, on écrit son nom suivi de parenthèses. Si la fonction admet des paramètres, leurs valeurs sont passées dans l'ordre, entre les parenthèses.

Si la fonction retourne une valeur, celle-ci peut être affectée dans une variable, servir directement dans une expression ou bien être ignorée :

```
int x;
x=somme(3,b); // additionne 3 et b
y=10+somme(5,7); // le résultat de la fonction est additionné à 10
somme(1,2);    // résultat perdu
```

Il est important de ne pas se poser trop de questions quant au fonctionnement interne d'une fonction à laquelle on fait appel : elle prend des paramètres et retourne une valeur. C'est suffisant pour l'appeler. Cette stratégie rend de grands services lors de la mise au point de fonctions récursives. Il s'agit de fonctions qui se rappellent elles-mêmes jusqu'à obtention d'un résultat déterminé.

d. Gestion des variables locales

Lorsqu'une fonction est appelée, elle construit un environnement local dans la pile. Cet environnement contient les valeurs des paramètres, puis les variables locales. Dans le corps de la fonction, paramètres et variables locales ont la même portée. L'instruction return provoque la destruction de cet environnement, et si la fonction retourne une valeur, cette valeur est laissée sur la pile à l'attention de la fonction appelante.

Nous en déduisons que les variables locales perdent leur contenu à l'issue de l'exécution d'une fonction. Le langage C a proposé un mot clé spécial, static, qui rend persistantes les valeurs des variables locales. S'agissant d'une curiosité qui n'existe pas dans d'autres langages que le C++, cette forme est à utiliser le moins possible.

Il n'y a de toute façon pas de correspondance algorithmique. Donnons toutefois un exemple pour illustrer sa syntaxe :

```
void dernier_appel()
{
static int heure;
  printf("time stamp du dernier appel : %d",heure);
  heure=time();
}
```

Quoi qu'il en soit, la création puis la destruction d'un environnement local est un mécanisme normal. Sans lui, nous aurions du mal à écrire des fonctions récursives, fonctions qui permettent de traiter avec beaucoup d'élégance des problèmes qui peuvent être complexes à programmer dans des versions itératives.

e. Définir des fonctions homonymes (polymorphisme)

Des fonctions qui portent le même nom ? On peut en conclure qu'elles remplissent le même rôle. Créer plusieurs versions d'une même fonction, voilà leur raison d'être. C'est leur signature, c'est-à-dire la qualité et la quantité

de leurs arguments, qui les distinguera.

On peut ainsi créer deux versions de la fonction somme : une première qui prend deux nombres, une deuxième qui en prend trois.

```
int somme(int a,int b)
{
  return a+b;
}
int somme(int a,int b,int c)
{
  return a+b+c;
}
```

À l'appel de la fonction somme, le compilateur choisit la forme qui lui paraît convenir le mieux. S'il n'en trouve pas, il génère une erreur.

```
int p=somme(3,2);  // utilise la première forme
int q=somme(4,5,6); // utilise la seconde forme
```

Attention toutefois, le compilateur ne peut pas toujours distinguer la version à utiliser. Si nous ajoutons une troisième version utilisant deux nombres de type **short** :

```
int somme(short a,short b)
{
  return a+b;
}
```

L'appel d'une version somme recevant 10 et 11 doit faire hésiter le compilateur. Ces littérales d'entier sont aussi bien des int que des short. Certains compilateurs soulèvent un avertissement (warning), d'autres ignorent ce fait. On peut alors utiliser un opérateur de transtypage par coercition pour indiquer au compilateur quelle forme utiliser :

```
somme((short) 3,(short) 4); // utilise la forme avec des short
```

On désigne parfois le polymorphisme de fonction (l'existence sous plusieurs formes) sous le nom de surcharge. Quoi qu'il en soit, cette notion est totalement indépendante de la programmation orientée objet qui n'a pas besoin d'elle pour exister.

f. Fonctions à nombre variable d'arguments

Nous avons déjà rencontré une fonction à nombre variable d'arguments : **printf**. Cette fonction admet comme premier paramètre une chaîne de formatage, puis une série de valeurs destinées à être présentées par le biais de ces formateurs :

```
printf("%s %d %x","bonjour",34,32);
```

Dans cet exemple, **printf** admet quatre arguments. Le premier recense trois formateurs (**%s**, **%d** et **%x**), il est suivi par trois valeurs conformes au type indiqué par les formateurs :

%s	chaîne (char*)
%d	entier (décimal)
%x	entier (hexadécimal)

Vous pouvez définir vos propres fonctions à nombre variable d'arguments. Des macros spéciales permettent de

traiter la liste des arguments transmise à votre fonction.

```
#include <iostream.h>
#include <stdarg.h>

int somme(int n, ...)
{
  va_list ap;      // liste des paramètres
  va_start(ap,n);    // se placer après le dernier argument formel

  int i,s=0;
  while(n--)
  {
    i=va_arg(ap,int);  // récupérer l'argument suivant de type int
    s+=i;
  }
  va_end(ap);      // nettoyer la liste des arguments
  return s;
}

int main(int argc, char* argv[])
{
  cout << somme(4,1,2,3,4); // affiche le résultat
  return 0;
}
```

Dans cet exemple, la fonction **somme** reçoit au moins un paramètre, **n**, qui indique le nombre d'éléments à additionner. Le symbole spécial **...** dans la signature de la fonction indique qu'il s'agissait du dernier paramètre formel, c'est-à-dire du dernier paramètre nommé. Les autres paramètres - qui peuvent être omis à l'appel - sont accessibles via la macro **va_arg()**.

g. Donner des valeurs par défaut aux arguments

Cette syntaxe n'a pas de correspondance algorithmique, mais elle peut guider le programmeur lorsqu'il hésite à fournir certains paramètres lors de l'appel d'une fonction.

Par exemple, imaginons la fonction suivante :

```
void printlog(char*message,FILE* f=NULL)
{
if(f==NULL)
  printf(message); // affiche à l'écran
else
  fprintf(f,message); // affiche dans un fichier
}
```

Le programmeur qui utilise notre fonction printlog comprend que la fourniture du paramètre f n'est pas obligatoire, puisqu'il a reçu une valeur par défaut.

On peut alors appeler **printlog()** de deux façons :

```
printlog("démarrage de l'application");
printlog("démarrage de l'application",f_erreur);
```

Dans le premier cas, le message s'affichera à l'écran, la fonction détectant une valeur **NULL** pour le paramètre **f**. Dans le second cas, **f_erreur** étant réputé non nul, le message s'inscrira dans un fichier préalablement ouvert, représenté par **f_erreur**.

Attention de ne pas provoquer de conflit entre les versions polymorphes (surchargées) et les versions de fonctions recevant des valeurs par défaut. La construction suivante est par exemple illicite :

```
int somme(int a,int b)
{
  return a+b;
}
int somme(int a,int b,int c=0)
{
  return a+b+c;
}
```

À l'appel de la fonction **somme** recevant deux arguments, le compilateur ne pourra déterminer s'il s'agit de l'omission du paramètre c ou bien si le programmeur a l'intention d'utiliser la première forme.

h. Fonctions en ligne

Les fonctions en ligne offrent un temps d'appel très court puisque précisément elles ne provoquent pas de débranchement (go sub). Le code qu'elles renferment est développé en lieu et place de l'appel.

À l'utilisation, cette caractéristique n'apparaît pas dans la syntaxe mais cela peut faire croître la taille du code de manière inopportune si la fonction est appelée en différents points du programme.

```
inline int somme(int a,int b)
{
    return a+b;
}

...
x = somme(3,4); // Place le code ici plutôt que d'effectuer
un débranchement
```

i. Fonctions externes de type C

La directive **extern "C"** indique au compilateur qu'il doit utiliser une convention d'appel de type C pour appeler une fonction. Les langages C++ et C ont des fonctionnements internes proches mais pas complètement identiques, notamment en ce qui concerne l'appel de fonction. Dans le cas du langage C, les paramètres sont empilés du dernier au premier, la fonction s'exécute puis l'appelant restaure la pile après avoir récupéré la valeur de retour de la fonction. Dans le cas du langage C++, l'ordre des paramètres est inverse et c'est l'appelé qui restaure son cadre de pile.

En conclusion, vous devez préfixer vos déclarations de fonctions par **extern "C"** si elles sont issues d'un compilateur C :

```
extern "C" int yylex();
```

j. Fonctions récursives

Il ne s'agit pas d'une spécificité du langage C++, aucune syntaxe particulière n'est nécessaire, mais plutôt d'une caractéristique supportée. Les fonctions C++ ont la possibilité de se rappeler elles-mêmes. Pour le lecteur qui découvre ce style de programmation, l'exemple du calcul de la factorielle est un bon point de départ.

La factorielle (notée en mathématique !, mais cette notation n'a rien à voir avec l'opérateur de négation booléenne du C++) est une "fonction" qui se détermine comme suit :

```
!1 = 1 = 1
!2 = 2 x 1 = 2
!3 = 3 x 2 x 1 = 6
!4 = 4 x 3 x 2 x 1 = 24
...
!n = n x (n-1) x (n-2) x ... x 1
```

Il est facile de donner une version dite itérative d'une fonction qui calcule cette factorielle. Le type **long** a été retenu car la factorielle croit très rapidement et la limite des deux milliards du type **int** est vite atteinte.

```
long factorielle(long n)
{
long r=1;
  while(n-- >0)
    r=r*n;  // on aurait pu noter aussi r*=n
  return r;
}
```

Cette version fonctionne parfaitement, si ce n'est qu'en l'absence d'un nom explicite pour la fonction, on aurait du mal à déterminer si elle calcule la factorielle ou un autre produit.

En reprenant l'expression générale de la factorielle, on peut procéder à une réécriture très simple :

```
!n = n x (n-1) x (n-2) x...x1
= n x !(n-1)
```

Autrement dit, la factorielle de n est égale à n multiplié par la factorielle de (n-1), avec comme point de départ ! 1=1.

Nous en déduisons une nouvelle version :

```
long factorielle_r(long n)
{
  if(n==1)
    return 1;  // !1=1
  else
    return n*factorielle_r(n-1); //  nx !(n-1)
}
```

L'écriture est beaucoup plus simple à comprendre et à reconnaître. Ceci dit, certains algorithmes se prêtent bien à ce style de programmation, comme le parcours de documents XML, alors que d'autres n'en tireront aucun profit. Par ailleurs, certains algorithmes deviennent vite voraces en termes d'espace utilisé par la pile.

Le calcul de la suite de Fibonnacci **fib(n)=fib(n-1)+fib(n-2)**, avec **fib(1)=fib(2)=1**, plante souvent aux environs de **fib(50)** tant le nombre de calculs en suspens est élevé.

k. La fonction main()

Tous les programmes C++ exécutables contiennent une fonction **main()**, appartenant à l'espace de noms global. Cette fonction peut parfois porter un nom un peu différent, cela dépend des compilateurs et des éditeurs de liens. En général, c'est main (principal).

La fonction **main()** renvoie en principe un code entier, la convention voulant qu'un code nul signifie que le programme a fonctionné normalement et qu'un code non nul indique une erreur dont l'interprétation est laissée aux soins du programmeur. Avec l'apparition des programmes graphiques (au détriment des utilitaires en ligne de commande), cette convention a un peu tendance à s'estomper. Cela dépend des systèmes d'exploitation. Certains compilateurs acceptent même une définition **void** pour main et renvoient un code 0 par défaut.

La fonction **main()** peut aussi admettre des paramètres destinés à recueillir les arguments passés sur la ligne de commande :

```
int main(int argc,char* argv[])
{
}
```

Le premier argument, de type entier, se nomme souvent argc - pour argument count. Il désigne le nombre d'arguments passés par la ligne de commande. En principe, il vaut au moins un, le tout premier argument étant

le nom du programme exécutable. Le second argument, argv - pour argument value - est un tableau de chaînes. On trouve également comme signature char** ce qui revient au même (voir la partie sur les pointeurs). Il est facile de prévoir une petite boucle pour afficher les paramètres de la ligne de commande :

```
/* affiche.cpp */
int main(int argc,char* argv[])
{
  for(int i=1; i<argc; i++)
    printf("argument n%d = %s\n",i,argv[i]);
}
```

Nous compilons ce programme avec la ligne suivante :

```
g++ affiche.cpp o affiche
```

Puis nous exécutons le programme :

```
affiche valeur1 valeur2 "salut les amis" 34
```

Le programme produit l'affichage suivant :

```
argument n1 = valeur1
argument n2 = valeur2
argument n3 = "salut les amis"
argument n4 = 34
```

Si certains arguments représentent des nombres, il faudra les convertir depuis le type chaîne dans le type voulu à l'aide des fonctions correspondantes, telles **`atoi()`** - alpha to integer - ou **`atof`** - alpha to float.

En résumé, les signatures suivantes sont possibles pour main() :

`void main()`	En fait int main(), le compilateur transformant lui-même la fonction et ajoutant un return 0 à la fin.
`int main()`	Version habituelle.
`int main(int argc)`	Licite mais sans intérêt.
`int main(char*argv[])` ou **`int main(char**argv)`**	Le programmeur doit bien contrôler le nombre d'arguments passés.
`int main(int argc,char*argv[])` ou **`int main(int argc,char** argv)`**	Version la plus logique lorsque l'on souhaite recueillir les arguments de la ligne de commande.

Quoi qu'il en soit, cette fonction **`main()`** est unique dans un programme. Les programmes qui n'en contiennent pas sont de ce fait destinés à construire des librairies statiques. Il est courant, pour les librairies dynamiques (DLL) de posséder une fonction **`libmain()`** chargée de procéder à des initialisations.

Enfin, selon les systèmes d'exploitation, on pourra trouver des signatures un peu différentes. C'est notamment le cas des applications graphiques Windows.

6. Les pointeurs

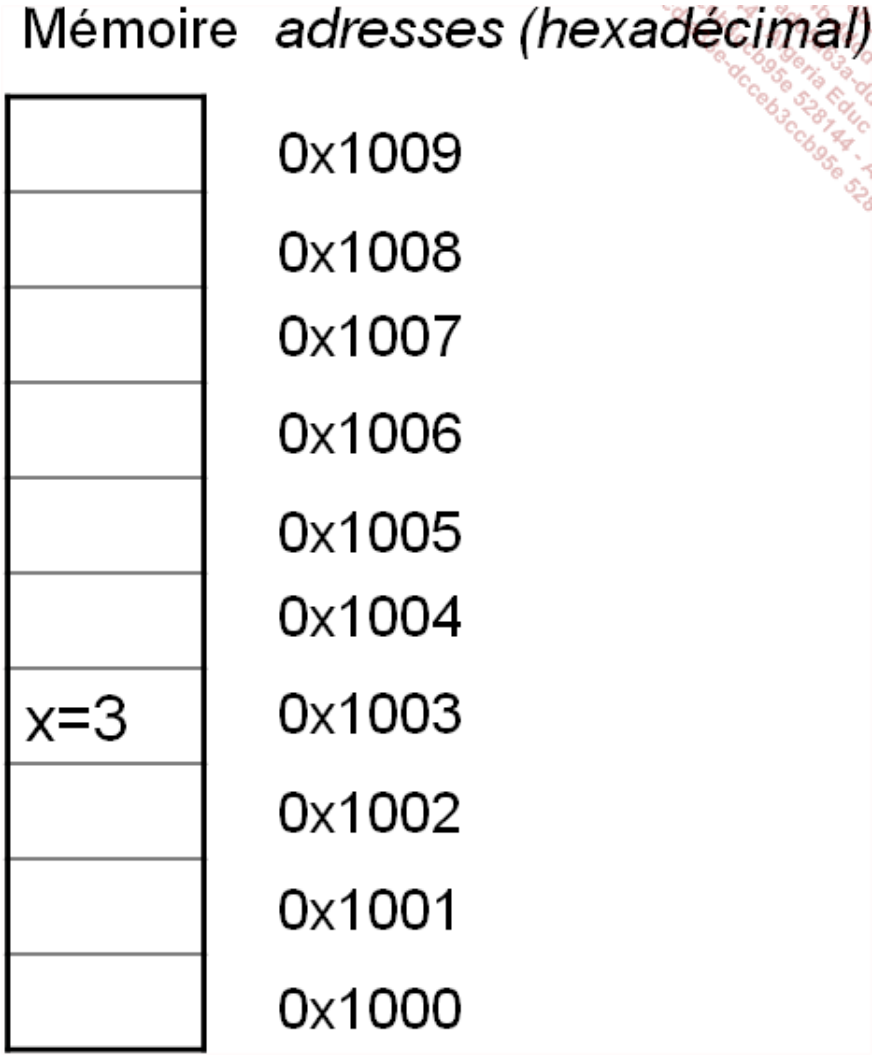
Les pointeurs et les références sont des outils particulièrement intéressants. Le langage C ne connaît pas les références, mais il peut travailler avec les pointeurs en suivant les règles applicables aux références. Des langages plus récents, comme Java, ont supprimé les pointeurs de leur vocabulaire. Non pas parce qu'ils

pourraient avoir mauvaise réputation auprès des programmeurs, mais parce qu'ils agissent à un niveau plus bas que les références, ce qui perturbe l'usage d'outils de haut niveau tels que le ramasse-miettes (garbage collector).

Pointeurs et références sont des variables qui permettent d'atteindre d'autres variables. Pour parvenir à ce résultat, le pointeur (ou la référence) utilise l'adresse de la variable cible, c'est-à-dire le numéro de la case mémoire où est rangée la valeur de cette variable. Comme la mémoire est comptée en octets, et qu'une variable peut répartir la représentation de sa valeur sur plusieurs octets, pointeurs et références sont des variables définies pour travailler avec un type donné, dans le but de limiter les erreurs d'adressage.

a. Pointeurs sur des variables

Commençons par étudier la représentation d'un fragment de la mémoire de l'ordinateur. Ce fragment contient une variable x de type char, préalablement initialisée à la valeur 3. Le plus souvent, les adresses s'écrivent en hexadécimal pour mieux les distinguer des valeurs stockées en mémoire, mais aussi car les adresses 16 ou 32 bits s'écrivent facilement dans cette base.



Pour affecter la valeur 10 à la variable x, par exemple, nous pouvons utiliser l'extrait de code suivant :

```
char x=3;
     x=10;
```

Si nous pouvions obtenir l'adresse de la variable **x**, 0x1003 dans notre cas, nous pourrions modifier cette variable sans utiliser directement x. Pour cela, nous allons définir un pointeur de type char, noté char*. Cette variable spéciale, **p**, recevra l'adresse de la variable **x**, déterminée à l'aide d'une syntaxe spéciale.

Ensuite, nous pourrons modifier la valeur située à cet emplacement mémoire, même si la variable **x** n'est plus dans notre portée.

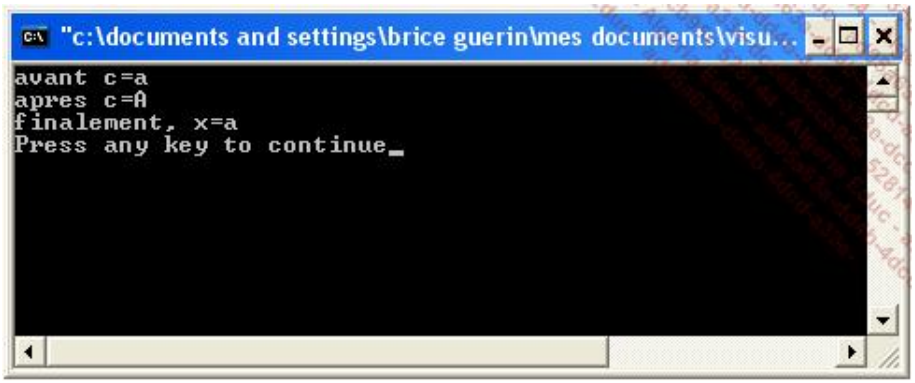
```
char* p;  // déclare un pointeur de type char

  p=&x;  // obtient l'adresse de la variable x

  printf("p=%x",p);  // affiche 1003 en hexa

  *p=10;  // affecte indirectement la variable x
```

Il faut noter l'aspect quelque peu artificiel de cet exemple. Pour modifier la valeur d'une variable, la syntaxe habituelle convient très bien et il n'y a pas besoin d'en changer.

Pour comprendre l'utilité de cette approche, créons une procédure qui transforme un caractère minuscule en majuscule :

```
void maj(char c)
{
  printf("avant c=%c\n",c);
  if(c>='a' && c<='z')
    c=c-('a'-'A');
  printf("après c=%c\n",c);
}

int main()
{
  char x='a';
  maj(x);
  printf("finalement, x=%c\n",x);
  return 0;
}
```

Toutefois, l'exécution de ce programme ne donne pas les résultats attendus :



Que s'est-il passé ? À l'appel de la fonction **maj()**, nous avons transmis par l'intermédiaire de la pile une copie de la variable **x**. Dans la portée de la fonction **maj()**, cette valeur s'appelle c. Il s'agit d'un paramètre qui a la durée de vie d'une variable locale. Tout se passe comme si nous avions écrit :

Fonction maj	Fonction main
`char c=x`	`maj(x)`
`printf("avant c=%c\n",c);` `if(c>='a' && c<='z')`	

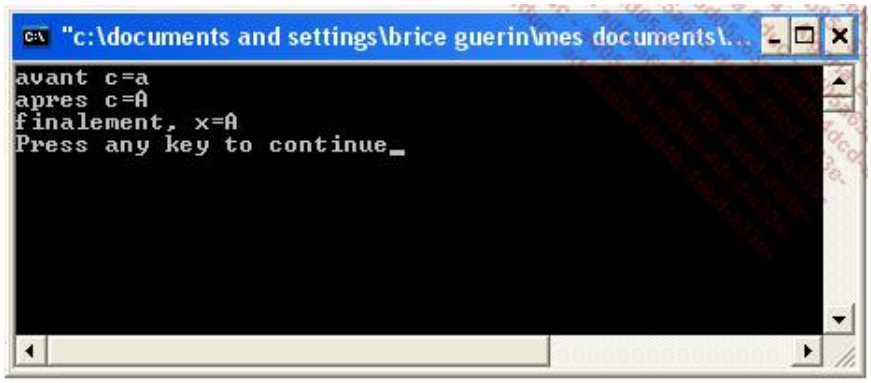
`c=c-('a'-'A');` `printf("après c=%c\n",c);`	
	`printf("finalement, x=%c\n",x);`

On comprend alors que la variable **x** est restée bien tranquille ! C'est sa copie, **c**, qui a été modifiée.

À présent, nous modifions la fonction **maj()** pour qu'elle reçoive non une valeur de type char mais un pointeur vers une variable de type char :

```
void maj(char* c)
{
  printf("avant c=%c\n",*c);
  if(*c>='a' && *c<='z')
    *c=*c-('a'-'A');
  printf("après c=%c\n",*c);
}

int main()
{
  char x='a';
  maj(&x);
  printf("finalement, x=%c\n",x);
  return 0;
}
```

L'exécution est cette fois-ci conforme à nos attentes, la variable **x** a bien été modifiée :



Il est temps de résumer les notations relatives aux pointeurs :

char* p	déclare p comme pointeur de type char, c'est-à-dire comme pointeur sur une variable de type char.
&x	désigne l'adresse de la variable x.
***p**	si p est un pointeur, désigne la valeur pointée par p, donc la valeur située à la case mémoire indiquée par p.

Maintenant que **maj()** admet un pointeur sur **char**, il n'est plus possible de l'appeler en lui transmettant une littérale de caractère :

Signature	Appel	Commentaire

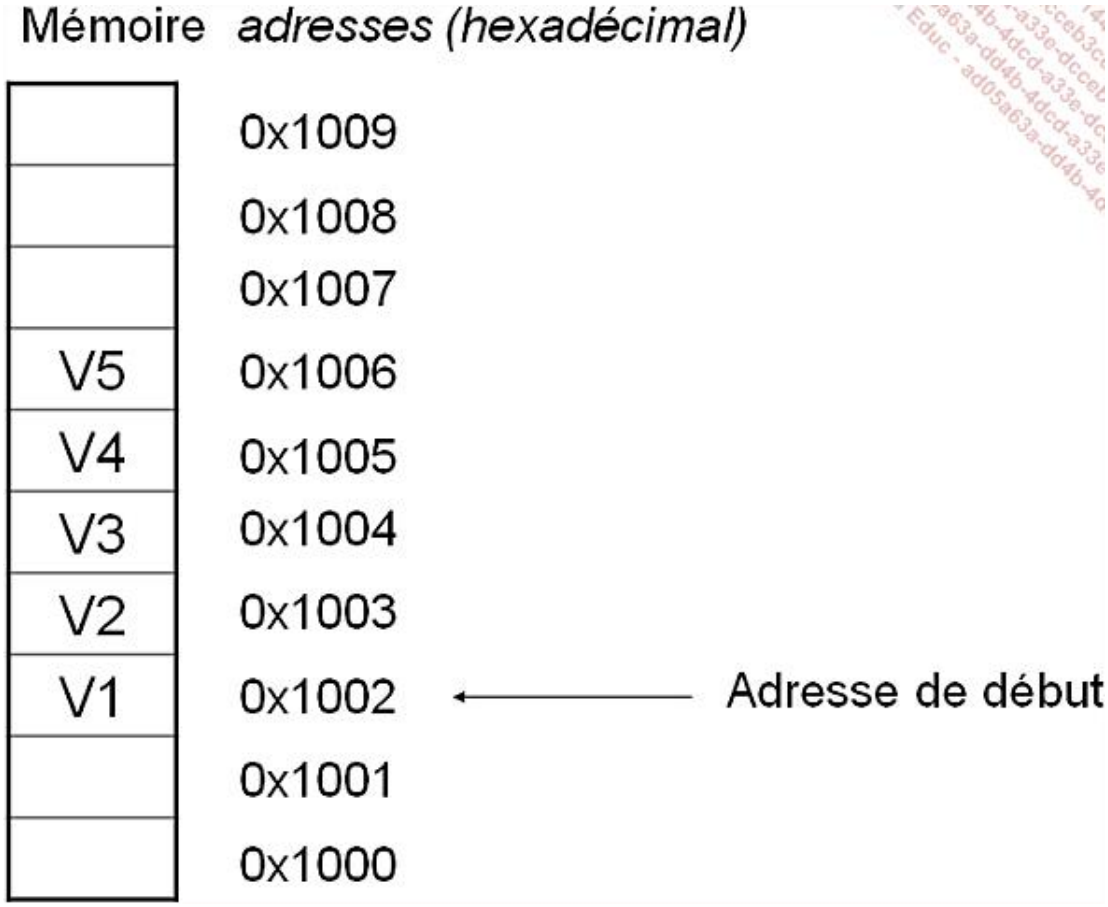
void maj(char c)	**maj(x)**	c initialisé à la valeur de x
void maj(char c)	**maj('t')**	c initialisé à la valeur 't'
void maj(char* c)	**maj(&x)**	c pointeur sur char, désigne la variable x
void maj(char* c)	**maj(&'t')**	erreur, une littérale de char n'a pas d'adresse

b. Pointeurs et tableaux

Nous avons vu précédemment la syntaxe de déclaration d'un pointeur sur une variable. Ce pointeur sert à atteindre une case mémoire par l'intermédiaire de la notation * :

```
char* p;
char c;
  p=&c;
  *p='K';
```

Imaginons que la cible ne soit plus une variable c, mais une plage de caractères (autrement dit, un tableau de char). Nous obtiendrons la représentation mémoire suivante pour une plage de 5 valeurs débutant à l'adresse 0x1002 :



Pour déclarer un tableau de 5 char, nous pouvons utiliser la syntaxe suivante :

```
char tab[]={'S','A','L','U','T' };
```

Le compilateur va ranger ces cinq valeurs dans une partie de la mémoire. La variable **tab** contiendra en fait l'adresse de début de cette plage. Autrement dit, **tab** se comporte comme un pointeur de char, en désignant le

premier de ces char.

Poursuivons notre raisonnement :

```
char* pt=tab;
```

Le pointeur de **char**, **pt**, a reçu l'adresse du tableau, c'est-à-dire qu'il pointe sur le premier **char**. Nous en déduisons que les écritures suivantes sont équivalentes :

```
*pt='S';
tab[0]='S';
```

Maintenant, nous voudrions atteindre la deuxième case du tableau :

```
tab[1]='A';
```

En passant par le pointeur, nous avons trois moyens d'obtenir le même résultat :

Déplacement du pointeur	Notation tableau	Notation pointeur
`pt++; // désigne la case d'après *pt='A';`	`pt[1]='A';`	`*(pt+1)='A';`

La première notation consiste à déplacer le pointeur. Rappelons-nous qu'il s'agit d'une variable, dont la valeur entière est une adresse. Incrémenter cette valeur d'une unité revient à déplacer le pointeur afin qu'il désigne la case voisine. Ensuite, nous utilisons la notation habituelle, ***pt**, pour écrire à cette position de la mémoire.

La deuxième notation, **pt[1]**, pousse encore la similitude entre pointeur et tableau. Si pt et tab coïncident sur la première case, la notation à base de crochets **[ ]** doit également correspondre pour toutes les valeurs du tableau.

Pour expliquer la troisième notation, commençons par étudier l'expression suivante :

```
*pt='A';
```

Puisque **pt** représente une adresse (la valeur de la variable), il semble logique d'écrire :

```
*(pt)='A';
```

Maintenant, considérant que pt est un nombre entier, il est possible de lui additionner un autre entier, la somme des deux représentant une nouvelle adresse :

```
*(pt+1)='A';
```

Si pointeurs et tableaux sont si proches, pourquoi conserver les deux ? Il faut considérer que le pointeur est une variable totalement libre, étant affectée avec l'adresse d'une variable **&v**, ou bien recevant l'adresse d'un bloc mémoire fraîchement alloué par l'opérateur **new** ou par la fonction **malloc()**. Le tableau lui peut s'initialiser avec des valeurs en extension ou bien avec l'opérateur **new**, mais pas avec la fonction **malloc()**.

Ensuite, les pointeurs sont employés lorsque l'on a recours à l'arithmétique des pointeurs ; ils sont prévus pour être déplacés au gré des nécessités de l'algorithme. Le tableau lui est fixe, et la notation suivante est prohibée :

```
char tab[]={ 2,3,44 };
```

```
tab++; // interdit
```

Enfin, les pointeurs sont utiles à certains algorithmes seulement. Privilégiez les tableaux chaque fois que ce sera possible, votre programme sera nettement plus portable. D'autant que les références constituent une bonne alternative aux pointeurs. Mais à l'époque de la création du langage C, la programmation était de beaucoup plus bas niveau que maintenant. D'autre part, bien maniés, les pointeurs se révèlent plus rapides que les références, ce qui est important dans certaines situations.

c. Allocation de mémoire

Nous l'avons vu, le langage C++ a conservé les mécanismes du langage C, tout en cherchant à en améliorer certains. La mémoire fait partie de ceux-là.

Il existe deux façons de réserver de la mémoire : en demandant au système, ou bien en utilisant des instructions du langage. La différence est subtile, mais cruciale.

Dans l'approche système, mise au point par le langage C, nous disposons d'une fonction **`malloc()`** chargée de réserver des octets. Cette fonction demande au système d'allouer une plage de n octets, que l'on interprétera par la suite comme étant une plage de valeurs d'un type donné, au moyen d'un transtypage. Comme c'est le système qui a alloué cette mémoire, il est également nécessaire de la lui rendre au moyen de la fonction **`free()`** qui admet un pointeur sur **`void`**, autrement dit, un pointeur de n'importe quel type.

Lorsque l'on utilise les instructions **`new`** et **`delete`**, le langage adapte sa gestion de la mémoire en fonction du type réservé. Aucun transtypage n'est nécessaire, ce qui simplifie la syntaxe.

Dans les faits, avec les compilateurs modernes, la fonction malloc() et l'instruction new partagent le même espace mémoire, appelé le tas, pour effectuer leurs réservations. Toutefois, les gestionnaires d'allocations n'étant pas les mêmes, il faut veiller à désallouer la mémoire avec le moyen qui correspond : **`free()`** pour rendre la mémoire allouée par **`malloc()`**, **`delete()`** pour rendre la mémoire obtenue par **`new`**.

Si vous développez un nouveau programme C++, privilégiez l'instruction **`new`**. La fonction **`malloc()`** doit être réservée à la portabilité des anciens programmes C. Ceci dit, il existe des cas où il faut employer une fonction pour allouer de la mémoire ayant un accès partagé : le Presse-papiers sous Windows, une zone de mémoire partagée sous Unix...

Allocation par malloc()

La fonction **`malloc()`** est déclarée dans l'en-tête **`<memory.h>`**. D'autres en-têtes peuvent convenir, comme **`<stdlib.h>`**.

Cette fonction est déclarée selon le prototype suivant :

```
void* malloc(int n);
```

La fonction est chargée d'allouer n octets dans la mémoire du système (en fait, celle du processus). Elle retourne l'adresse de cette plage sous la forme d'un pointeur sur void.

Si la réservation ne peut être satisfaite, la fonction renvoie 0, l'adresse 0, qui est considérée comme inaccessible. Pour éviter d'employer une valeur littérale aussi évocatrice, les concepteurs du langage C ont imaginé la macro NULL :NULL

```
#define NULL ((void*)0)
```

Si la réservation est satisfaite, le programmeur doit convertir le pointeur void* dans un type approprié :

```
char*p;
  p=(char*) malloc(15); // alloue 15 char
  if(p==NULL)
```

```
    printf("erreur d'allocation");
```

Nombre de compilateurs accepteraient l'écriture **p=malloc(15)**, mais par rigueur d'écriture, on doit s'efforcer d'effectuer un transtypage par coercition, en indiquant que l'on prend la responsabilité d'interpréter cette zone comme étant une zone de 15 char.

Par ailleurs, si le type est plus large que le char, il faut employer l'opérateur **sizeof()** pour réserver suffisamment d'octets :

```
double*d=(double*) malloc(30*sizeof(double)); // alloue 30 double
```

Lorsque le bloc est alloué, on s'en sert comme n'importe quel tableau, en utilisant la notation de son choix : *d, d[] ou *(d+i).

Lorsque la mémoire n'est plus utile, le bloc doit être rendu au système par l'intermédiaire de la fonction free() :

```
free(d);
```

Allocation par new

L'allocation par **new** sera pour l'instant réservée aux tableaux. Lorsque nous traiterons l'instanciation, l'opérateur **new** accomplira un autre rôle, essentiel.

La syntaxe générale de l'instruction **new** est :

```
type* new type < [taille] >
```

Autrement dit, l'opérateur new renvoie le pointeur vers le type alloué. La taille est optionnelle, l'unité est la valeur par défaut.

Voici quelques exemples d'allocation par l'opérateur new :

`char*p;` `p=new char[15];`	Allouer 15 char
`double*d;` `d=new double[30];`	Allouer 30 double
`string * s;` `s=new string;`	Allouer une chaîne à l'aide de la classe string (cf. Instanciation de classes)

Une fois le bloc alloué par **new**, on l'utilise avec les mêmes notations que lorsqu'il a été réservé par **malloc()**. En revanche, le bloc doit être impérativement libéré à l'aide de l'instruction **delete** :

```
delete(p);
```

d. Arithmétique des pointeurs

Lorsque nous déplaçons un pointeur, par incrément ou par addition, combien d'octets sont balayés ? Si le pointeur est char*, la réponse est simple :

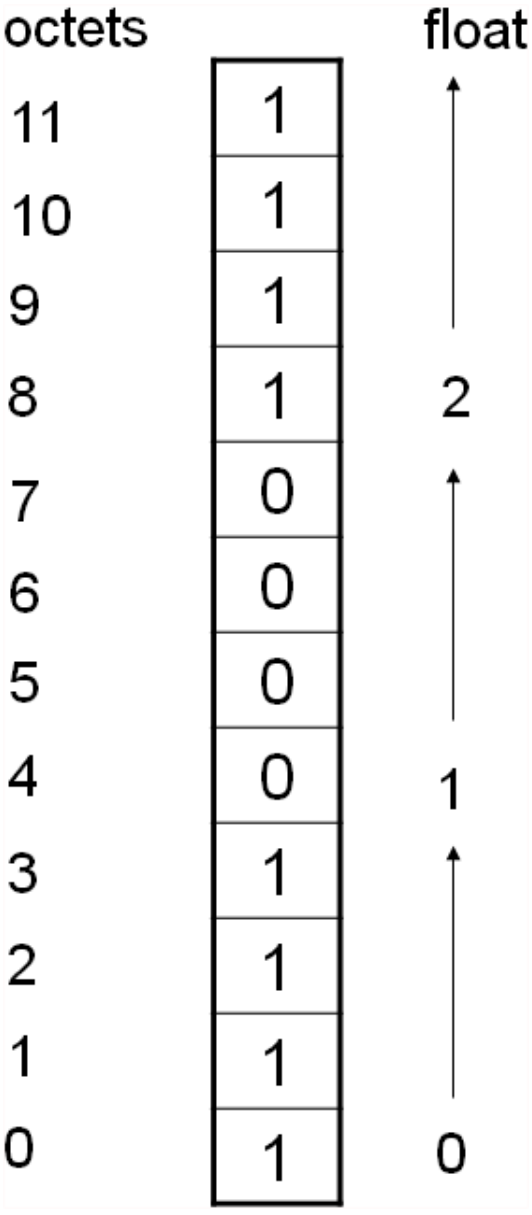
```
char*p;
char c;
     p=&x;
     p++;  // passe à l'adresse suivante (en octets)
```

De même, lorsque l'on accède à la valeur désignée par un pointeur, combien d'octets sont lus ou écrits ? Encore une fois cela dépend du type du pointeur.

Prenons l'exemple suivant :

```
float*p;
char*t;
    t=new char[12];
    for(int i=0; i<10; i++)
     t[i]=1;
    p=(float*)t; // peu rigoureux mais légal
    *(p+1)=0;
```

En sortie de ce programme, quel est l'état du tableau t ? L'écriture *(p+1)=0 a impacté 4 octets, à compter de la cinquième position, comme le montre l'illustration ci-après :



Lorsque l'on compte en **float**, il faut multiplier par 4 tous les déplacements. L'écriture impacte aussi 4 octets au lieu d'un seul, et ce, même si la zone avait été réservée comme une zone de char.

Cette règle de calcul s'appelle l'arithmétique des pointeurs, et doit être appliquée avec la plus grande rigueur qui soit.

e. Pointeurs de pointeurs

Maintenant que nous connaissons bien les pointeurs, pourquoi ne pas définir un pointeur qui désigne une variable de type pointeur ? Cette opération est finalement assez courante, si l'on considère que les littérales de chaînes sont des tableaux de **char**, autrement dit des pointeurs de **char (char*)**. La fonction **main(),** admettant comme paramètre un tableau de chaînes, reçoit en réalité un pointeur de pointeurs.

Les pointeurs de pointeurs ne sont finalement pas si complexes, ils constituent simplement une indirection de plus. Il ne faut pas en abuser et, afin de simplifier au maximum les notations, on privilégiera les notations de type tableau.

```
char**argv;  // un pointeur de pointeurs de type char
char* argv[]; // un tableau de pointeurs de type char
```

Bien entendu, les notations destinées aux pointeurs usuels restent applicables aux pointeurs de pointeurs :

```
char*p;          // un pointeur de type char
char* *pp;       // un pointeur de pointeur de type char

pp=&p;           // &p=adresse de p. pp désigne p

*pp="salut";     // autrement dit, p="salut"
**pp='S';        // autrement dit, *p='S'

printf("%s",p);  // affiche Salut
```

f. Pointeurs de fonctions

Puisque les instructions définissant une fonction sont, à l'instar des variables, rangées dans la mémoire, nous pouvons admettre que les fonctions sont en fait des adresses : celles de leur première instruction. Partant de là, il devient possible de définir des pointeurs de fonctions pour appeler - indirectement - certaines d'entre elles.

Il apparaît que les pointeurs de fonctions sont employés dans quelques situations particulières. Tout d'abord, un pointeur de fonction rend certains algorithmes génériques. Prenons l'exemple de l'algorithme de tri rapide. Celui-ci reste le même, que l'on cherche à trier un ensemble d'entiers, un ensemble de booléens, ou des objets de nature variée. Pour rendre le programme indépendant du type de données à trier, il est possible d'utiliser un pointeur vers une fonction qui admette deux valeurs et qui indique laquelle est la plus grande.

On rencontre également des pointeurs de fonctions lorsque l'on applique des méthodes à des objets. Le chapitre sur l'adressage relatif nous donnera plus d'informations à ce sujet.

Enfin, il n'est pas rare de fournir à un module "système" un pointeur vers une fonction qui sera appelée lorsque surviendra un événement particulier. On désigne ce mécanisme par le terme de fonctions callback (rappelables).

Utilisation de pointeurs de fonctions pour rendre les algorithmes génériques

Dans le but d'illustrer cette approche, nous proposons d'étudier l'algorithme du tri rapide.

Pour comprendre cet algorithme, nous commençons par partitionner un tableau. Un tableau de valeurs, par exemple des entiers, est partitionné autour d'une valeur pivot en plaçant à gauche de ce pivot toutes les valeurs inférieures et à droite, toutes les valeurs supérieures.

Tout d'abord le tableau d'origine, avec un pivot (en gras), choisi arbitrairement au milieu :

2	8	3	7	**5**	9	3	10	3

Voilà maintenant le tableau partitionné. Le pivot peut avoir changé de place.

2	3	3	**5**	7	9	8	10	3

L'algorithme du tri rapide réitère cette partition, à gauche et à droite du pivot, récursivement. Finalement, nous récupérons un tableau complètement trié.

Voici pour commencer une implémentation fonctionnant pour un tableau d'entiers. Nous vous encourageons à la tester telle quelle si vous n'êtes pas familiarisé avec cet algorithme :

```
// chapitre 1 partition.cpp : définit le point
// d'entrée pour l'application console.
//

int partition(int* T,int m,int d)
{
  // valeur pivot, variable d'échange
  int v,aux;
  int m1=m,d1=d;
  // initialisation
  v=T[m+(d-m)/2];

  // tant que les index ne se croisent pas
  while(m<d)
  {
    // rechercher une valeur inférieure à droite
    while(m<d && T[d]>v)
          d--;
    // rechercher une valeur supérieure à gauche
    while(m<d && T[m]<v)
          m++;
    if(m>=d)
         break;

    if(T[m] != T[d])
    {  // échange
       aux=T[d];
       T[d]=T[m];
       T[m]=aux;
    }
     else
      d--;
  }

  return m;
}

void tri_aux(int* T,int m,int d)
{
  if(m>=d)
    return;      // rien à trier

  int k=partition(T,m,d); // partitionne entre m et d
  tri_aux(T,m,k-1);    // tri à gauche
  tri_aux(T,k+1,d);    // tri à droite
}

void tri(int* T,int length)
{
```

```
  tri_aux(T,0,length-1);
}

void afficher(int T[],int m,int d)
{
  // affichage du tableau à chaque étape
  for(int i=m; i<=d; i++)
      printf("%d,",T[i]);
  printf("\n");
}

int main()
{
  int tab[]={ 5,1,7,2,8,4,9,13};
  tri(tab,8);
  afficher(tab,0,7);
  return 0;
}
```

Nous proposons maintenant une implémentation C++ du tri rapide, prenant en paramètres un tableau à trier, des indices nécessaires au fonctionnement de l'algorithme, plus deux pointeurs de fonction. L'un désigne une fonction de comparaison, l'autre une fonction d'échange.

Voici tout d'abord, la définition des pointeurs de fonction. Comme l'écriture est un peu lourde, on a souvent recours à la définition d'alias de type (typedef) :

```
typedef void (*pf_echange)(void*,int,int);
typedef int (*pf_comp)(void*,int,int);
```

Le premier modèle de fonction, **pf_echange**, caractérise la signature d'une fonction ne renvoyant rien et admettant un tableau de **void** (tout type de valeur, en fait), ainsi que deux entiers.

Le second modèle, **pf_comp**, admet les mêmes paramètres mais renvoie un entier, résultat de la comparaison entre deux valeurs. Dans les deux cas, les entiers admis comme paramètres sont les index des valeurs du tableau, à comparer ou à échanger, selon le cas.

Il faut maintenant implémenter deux fonctions respectant ces signatures :

```
void int_echange(void*t,int p1,int p2)
{
  int*T=(int*)t; // transtypage (cast)
  int aux=T[p1];
  T[p1]=T[p2];
  T[p2]=aux;
}

int int_compare(void*t,int p1,int p2)
{
  int v1,v2;
  int*T=(int*)t; // transtypage (cast)
  v1=(int) T[p1];
  v2=(int) T[p2];

  return v1-v2;
}
```

Jusqu'à présent, nous n'avons pas d'autres moyens d'être indépendant vis-à-vis des types que d'utiliser un pointeur sur **void**. Ce qui explique le transtypage un peu brutal, le **void*** étant promu en **int***.

Il ne nous reste plus qu'à aménager le programme existant pour travailler avec ces fonctions, par l'entremise de pointeurs :

```
// l'algorithme devenu général
```

```
int partition(void* T,int m,int d,pf_echange fswap,pf_comp fcomp)
{
  // valeur pivot, variable d'échange
  int pv;
  int m1=m,d1=d;
  // initialisation
  pv=m+(d-m)/2;  // position du pivot

  // tant que les index ne se croisent pas
  while(m<d)
  {
    // rechercher une valeur inférieure à droite
    while(m<d && (*fcomp)(T,d,pv)>0)
       d--;

    // rechercher une valeur supérieure à gauche
    while(m<d && (*fcomp)(T,m,pv)<0)
      m++;

    if(m>=d)
      break;

    if((*fcomp)(T,m,d)!=0)
    {  // échange
      (*fswap)(T,m,d);
    }
     else
      d--;
  }

  return m;
}

void tri_aux(int* T,int m,int d,pf_echange fswap,pf_comp fcomp)
{
  if(m>=d)
    return;      // rien à trier
  int k=partition(T,m,d,fswap,fcomp); // partitionne entre m et d
  tri_aux(T,m,k-1,fswap,fcomp);    // tri à gauche
  tri_aux(T,k+1,d,fswap,fcomp);    // tri à droite
}

void tri(int* T,int length,pf_echange fswap,pf_comp fcomp)
{
  tri_aux(T,0,length-1,fswap,fcomp);
}

void afficher(int T[],int m,int d)
{
  // affichage du tableau à chaque étape
  for(int i=m; i<=d; i++)
      printf("%d,",T[i]);
  printf("\n");
}

int main()
{
  int tab[]={ 5,1,7,2,8,4,9,13};
  tri(tab,8,&int_echange,&int_compare);
  afficher(tab,0,7);
  return 0;
}
```

Au passage, nous relevons que les notations associées aux pointeurs de variables restent applicables aux pointeurs de fonctions :

&int_echange	adresse de la fonction **int_echange()**

(*fcomp)(T,m,d)	Contenu à l'adresse **fcomp**, autrement dit la fonction désignée par **fcomp** est appelée avec les paramètres (**T**,**m**,**d**).

Fonctions callback

L'API (*Application Programming Interface*) standard du C++ propose peu de fonctions callback. Pour les systèmes graphiques, comme Windows, c'est au contraire monnaie courante. Les gestionnaires d'événements associés à un clic sur un bouton sont souvent des fonctions callback.

Dans ce type de situation, nous avons généralement un système d'enregistrement. Pour un événement donné - un clic sur un bouton par exemple -, une ou plusieurs fonctions sont enregistrées.

Lorsque l'événement survient, chacune de ces fonctions est appelée dans le but d'accomplir un travail spécifique - ouvrir une fenêtre, réaliser un calcul...

Nous pouvons simuler cette approche à l'aide d'un petit programme. Il est constitué d'une partie système et d'une partie application. La partie système possède un dispositif pour enregistrer des gestionnaires d'événements, ainsi qu'une fonction qui scrute le clavier. Lorsque celui-ci est sollicité, tous les gestionnaires (ce sont des fonctions) sont appelés avec un argument représentant la touche pressée.

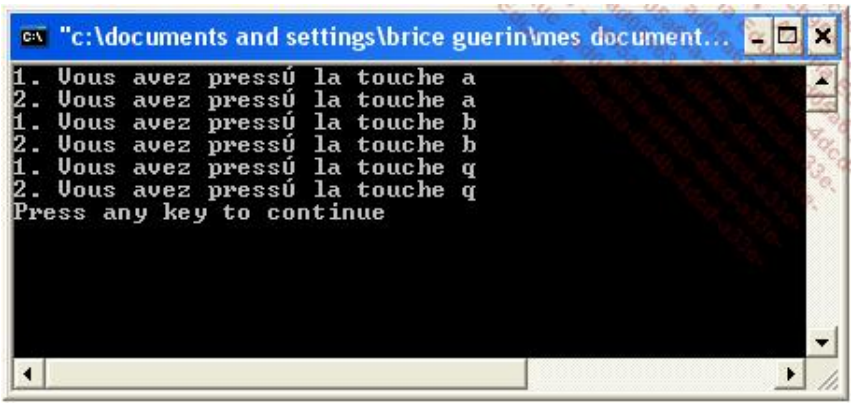
```
//  partie  système
#include  <conio.h>

typedef  void  (*pf_key_press)(int  key);

pf_key_press*gestionnaires;
int  nb_gestionnaires;

void  init()
{
    gestionnaires=new  pf_key_press[10];    //  10  gestionnaires  max
    nb_gestionnaires=0;
}

void  enregistrer(pf_key_press  gestionnaire)
{
    gestionnaires[nb_gestionnaires++]=gestionnaire;
}

void  propager_evenement(int  touche)
{
  for(int  i=0;i<nb_gestionnaires;  i++)
    (*gestionnaires[i])(touche);  //  appelle  le  gestionnaire
}

void  surveiller_clavier()
{
  int  touche;
  do
    {
    while(!_kbhit());  //  attend  la  frappe  d'une  touche
    touche=_getch();    //  récupère  la  touche

    propager_evenement(touche);
    }  while(touche!='q');
}
```

Voilà maintenant la partie application, qui consiste en deux gestionnaires plus une fonction main() :

```
// partie application
void touche_pressee1(int key)
{
  printf("1. Vous avez pressé la touche %c\n",key);
```

```
}

void touche_pressee2(int key)
{
  printf("2. Vous avez pressé la touche %c\n",key);
}

// main()
int main(int argc, char* argv[])
{
  // initialisation du système
  init();

  // ces deux fonctions seront appelées lorsque le clavier
  // sera sollicité
  enregistrer(&touche_pressee1);
  enregistrer(&touche_pressee2);

  // démarrage
  surveiller_clavier();
  return 0;
}
```

L'exécution du programme peut donner quelque chose qui ressemble à cela :



Bien que sommaire, ce programme illustre bien le fonctionnement des systèmes d'exploitation graphiques comme Windows ou Mac OS.

7. Références

Bien qu'accomplissant le même rôle que les pointeurs, les références offrent une syntaxe plus simple et en même temps limitent les risques d'accès erroné à la mémoire.

Une référence est toujours associée à une variable, alors qu'un pointeur peut être modifié via l'arithmétique des pointeurs.

La syntaxe de définition d'un type référence utilise le préfixe &, en remplacement de l'étoile. Par contre, il ne faut pas confondre ce préfixe avec l'opérateur & qui extrait l'adresse d'une variable ou d'une fonction.

```
char c;  // un caractère
char*p;  // un pointeur de char
p=&c;    // p désigne c, et &c représente l'adresse de c
char & refc=c; // refc est une référence de char,
refc désigne c
```

Dans cet extrait de code, nous avons défini une référence de char, refc, désignant la variable c. Cette référence est devenue un alias de la variable c, aussi, toute modification amenée par refc impactera en réalité c, même en

dehors de sa portée :

```
refc++;  // incrémente en fait c
```

Comme dans le cas des pointeurs, les références n'ont de réel intérêt que pour réaliser des effets de bord. Pour illustrer cet usage, reprenons la fonction maj() étudiée précédemment.

Version pointeur	Version référence

`void maj(char* c)`
`{`
`  printf("avant c=%c\n",*c);`
`  if(*c>='a' && *c<='z')`
`  *c=*c-('a'-'A');`
`  printf("après c=%c\n",*c);`
`}`
	`void maj(char& c)`
	`{`
	`  printf("avant c=%c\n",c);`
	`  if(c>='a' && c<='z')`
	`  c=c-('a'-'A');`
	`  printf("après c=%c\n",&c);`
	`}`

`int main()`
`{`
`char x='a';`
`  maj(&x);`
`  printf("finalement,`
`x=%c\n",x);`
`  return 0;`
`}`
	`int main()`
	`{`
	`char x='a';`
	`  maj(x);`
	`  printf("finalement,`
	`x=%c\n",x);`
	`  return 0;`
	`}`

Nous nous rendons compte que la syntaxe par référence est plus simple que celle proposée par le style pointeur. L'écriture se rapproche davantage du passage par valeur et pourtant l'effet de bord - la modification par une procédure - est possible.

Une plus grande sécurité

Les références offrent une sécurité bien plus grande que les pointeurs, puisque l'adresse maniée par la référence n'est pas évaluable. Le pointeur est libre d'être déplacé, par incrément, addition... La référence étant toujours, dès sa déclaration, associée à une variable, les risques d'erreurs sont limités.

Les références constantes

Les références constantes, généralement utilisées pour les fonctions, garantissent que la variable n'est pas modifiée par la fonction.

```
void fonction_sans_risque(const int & x)
{
  printf("x=%d",x);  // ok
  x=4; // erreur
}
```

On peut objecter que l'intérêt d'une telle méthode est quasi nul. À quoi bon passer un entier par référence si l'on s'évertue à le rendre invariable ? Pour un type primitif comme **`int`**, la cause est entendue, mais pour un type objet (une classe), les cas d'application sont nombreux. Les méthodes sont applicables, les champs sont modifiables, même si la référence est constante. Le passage par référence autorise les effets de bord, l'objet n'étant pas copié dans la pile. D'autre part l'appel est plus rapide puisque la référence s'assimile à une adresse, ce qui est souvent plus économique que de recopier tous les champs de l'objet dans la pile.

Renvoyer une référence depuis une fonction

Il est tout à fait possible de définir une fonction qui renvoie une référence. En voici un exemple :

```
double euro,fs,livre;
double & cours(int pays)
{
  switch(pays)
  {
    case 1:
      return livre;
    case 2 :
      return fs;
    default:
      return euro;
  }
}
```

L'intérêt d'une telle approche est que tout le monde travaille avec la même valeur. Si les variables **euro**, **fs** ou **livre** voient leur valeur modifiée, toutes les fonctions les utilisant utiliseront les valeurs à jour, la fonction **cours()** ayant transmis une référence plutôt qu'une valeur.

Cet emploi des références évoque également l'usage de champs statiques dans une classe, avec certes une syntaxe un peu différente.

8. Constantes

a. Constantes symboliques

Le préprocesseur, utilitaire de prétraitement textuel, remplit trois missions importantes :

- il inclut les fichiers désignés par la directive **#include** ;
- il évalue la présence de macros par la directive **#ifdef** ;
- il évalue les macros définies par la directive **#define**.

Ce préprocesseur travaille en amont du compilateur. Depuis que les nouveaux langages de programmation ont abandonné son usage (Java par exemple), il vaut mieux limiter le nombre de macros définies avec **#define**.

On peut toutefois utiliser cette dernière directive pour définir des constantes symboliques :

```
#define PI 3.14159265358
```

Dans les fichiers qui ont reçu cette définition, le préprocesseur remplacera la chaîne PI par sa valeur textuelle dans toutes ses occurrences qui n'apparaissent pas dans une chaîne de caractères.

```
#define PI 3.14
double x=PI;

char*a="Le savant grec Pythagore a découvert PI
sans calculette";
```

Le compilateur reçoit une version modifiée du dernier fragment :

```
double x=3.14;

char*a="Le savant grec Pythagore a découvert PI
sans calculette";
```

b. Le type void

Aucune variable ne peut être typée void, pourtant ce type entre dans la classification des types élémentaires, comme **int**, **short**...

Ce mot clé est utilisé pour indiquer qu'une fonction ne renvoie rien, c'est une procédure :

```
void afficher(char*s)
{
}
```

On utilisera aussi **void *****, pointeur sur un type indéterminé. Un transtypage sert ensuite à convertir ce pointeur dans le type qui convient :

```
void* malloc(int nb_octets) // fonction qui renvoie void*
{ ..}

char*s=(char*) malloc(15) ; // alloue 15 octets
```

Ce système était pour le C un moindre mal dans sa tentative de supporter la généricité. Plutôt que d'assimiler l'adresse à un entier - ce qui se passe de toute façon lorsque l'on parle d'arithmétique des pointeurs - on dira que **malloc()** alloue des octets et retourne l'adresse de la zone allouée. Le type de la zone (int, double...) dépend de l'interprétation qu'en fait le programmeur. Cette interprétation est précisée par la conversion d'une valeur **void*** en **char*** dans notre exemple.

c. Les alias de type, typedef

Cet opérateur sert à créer des alias vers des types exigeant des notations alambiquées. Plutôt que d'écrire très souvent **unsigned char**, on préférera déclarer le type **uchar** :

```
typedef unsigned char uchar ;
```

Par la suite, des variables prendront indifféremment le type **unsigned char** ou **uchar** :

```
uchar c ; // équivalent à unsigned char c ;
```

d. Constantes et énumérations

Le langage C++ a introduit le mot clé **const** qui empêche la modification d'une variable après son initialisation :

```
const double pi=3.14;
```

Cette notation est commune à d'autres langages, ce qui la rend bien plus portable que la directive **#define**.

Lorsque la valeur numérique d'une constante importe moins que son libellé, les énumérations constituent une très bonne solution :

```
enum Jour { lundi, mardi, mercredi, jeudi,
vendredi, samedi, dimanche } ;
Jour rendez_vous;

int main(int argc, char* argv[])
{
  rendez_vous=mercredi;
  return 0;
}
```

Dans cette configuration, la valeur réelle de mercredi importe peu. C'est le nom de la constante qui est évocateur.

En fait, les énumérations sont assimilables à un type entier, aussi est-il possible de fixer la valeur de départ de l'énumération :

```
enum Couleur { Rouge=38, Jaune, Bleu };
```

Exceptions

1. Les approches de bas niveau

Tout programme est soumis aux aléas de l'environnement qui le fait fonctionner. Il peut intervenir des défaillances matérielles, certains processus bloquent des ressources critiques, la mémoire n'est pas inépuisable...

Pour qu'un programme entre dans la catégorie des logiciels, il doit être tolérant vis-à-vis de ces événements et ne pas se bloquer, ou pire, s'interrompre brutalement.

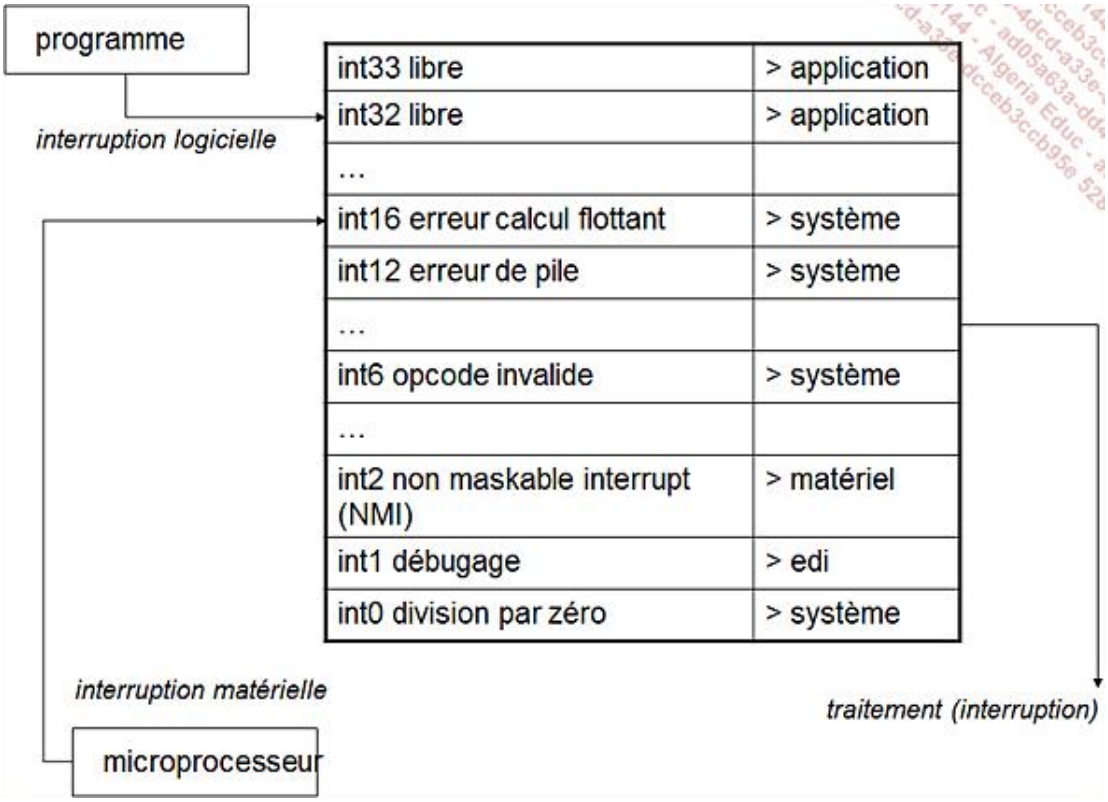
Les anciens langages de programmation sont en général assez mal pourvus pour traiter les situations problématiques, et les développeurs se sont souvent appuyés sur les dispositifs prévus par leur environnement.

a. Drapeaux et interruptions

Certains microprocesseurs disposent d'instructions destinées à déclencher des sous-programmes - des fonctions en langage C - lorsque certaines conditions sont réunies : une division par zéro, un dépassement de capacité, une faute dans la gestion de la mémoire paginée... Il n'est pas rare de voir une partie de ces interruptions laissées à l'entendement du programmeur système. Ainsi, le système d'exploitation MS-DOS a-t-il programmé sur l'interruption n°19 le redémarrage du système. Il existe des pilotes de périphériques qui emploient ces interruptions pour synchroniser des échanges de données. Citons les pilotes de disques durs, d'affichage et ceux prenant en charge les cartes d'acquisition numérique ou les cartes son.

À l'aide des instructions adéquates, qui peuvent être symbolisées par l'appel d'une fonction de l'API mise à disposition par le système d'exploitation, le programme enregistre l'adresse d'une fonction dite callback pour un numéro d'interruption donné.

Cette interruption peut ensuite être déclenchée directement par le microprocesseur - en cas de division par zéro par exemple - ou par un programme. Lorsque l'interruption est déclenchée, le microprocesseur appelle automatiquement la fonction callback. Lorsque c'est un programme qui est à l'origine du déclenchement de l'interruption, on parle d'interruption logicielle.



Pour autant, les interruptions ne constituent pas un moyen assez général pour le traitement des erreurs. Pour commencer, tous les microprocesseurs ne sont pas équipés de ce dispositif. D'autre part, l'unique table des interruptions est gérée par le système. Enfin, certaines interruptions doivent être déclenchées avec des priorités plus fortes que d'autres.

En résumé, les interruptions constituent une aide intéressante lorsque le programme décrit une couche du système d'exploitation, mais se révèlent trop limitées pour gérer des erreurs applicatives. Il est par contre fréquent que le gestionnaire d'interruptions prévu par le système d'exploitation avise le processus qui fonctionne au moment où survient l'incident au moyen d'une API de plus haut niveau.

De nombreuses fonctions de la bibliothèque standard du C adoptent un comportement particulier pour signaler qu'une opération n'a pas pu aboutir. Le plus souvent, le résultat envoyé a pour valeur un code indiquant l'état de l'opération. Ainsi la fonction **`getc(FILE*)`** renvoie un char sur 4 octets. Si l'entier vaut -1 (0xFFFFFF en hexadécimal), le flux est arrivé à épuisement. La fonction **`fopen()`** renvoie un pointeur NULL si l'ouverture du fichier n'a pas pu aboutir.

Le programmeur peut donc détecter ces situations finalement assez simples à traiter. Le plus souvent, un message sorti sur la console avertit l'utilisateur des circonstances supposées entourant la détection du problème.

Ce mécanisme trouve cependant ses limites ; dans le cas d'un algorithme sophistiqué, la fonction qui détecte un problème n'a peut-être pas les moyens de décider des actions à mener pour le résoudre. Ainsi, le développeur sera-t-il tenté d'utiliser une variable globale signalant la présence d'une erreur. Il n'est en effet pas opportun de modifier la signature des fonctions car cela reviendrait à remettre en cause l'algorithme.

La gestion des erreurs à base de drapeaux pose souvent des problèmes d'organisation, l'accès concurrentiel n'étant en général pas assuré, ni même le déroutement asynchrone lorsque le problème survient.

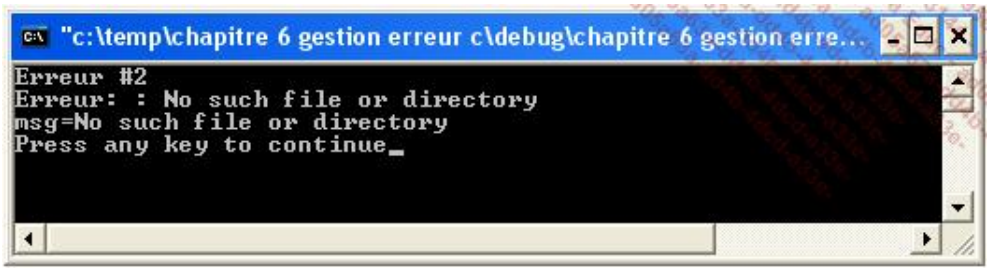
b. Traitement des erreurs en langage C

La bibliothèque du langage C met à disposition du programmeur des informations importantes sur la nature de l'erreur qui vient de se produire. Ainsi lorsqu'un appel à **`fopen()`** échoue, plusieurs raisons doivent être départagées : absence du fichier, droits d'accès insuffisants, avarie matérielle...

Le programme suivant utilise des fonctions du langage C tout à fait applicables en C++ pour expliquer, de différentes manières, les circonstances d'un **`fopen()`** ne fonctionnant pas :

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

int main(int argc, char* argv[])
{
  FILE* f;
  f = fopen("introuvable.txt", "rb");
  if(f==NULL)
  {
    // détection de la nature de l'erreur
    int e = errno;  // dernière erreurFonction;errno
    printf("Erreur #%d\n",e);

    // affichage d'un message circonstancié
    perror("Erreur: ");Fonction;perror

    // sortie d'un message sur un char*
    char* msg = strerror(errno);
    printf("msg=%s\n",msg);
  }
  else
    fclose(f);
  return 0;
}
```

Le fichier **introuvable.txt** étant évidemment absent de notre exemple, nous obtenons les résultats suivants :



Les fonctions offertes par la bibliothèque du langage C sont un premier pas mais ne résolvent pas tous les problèmes. Pour commencer, la variable **errno** est une variable globale, donc unique, et sa valeur risque d'être écrasée lorsqu'une autre erreur intervient si la consultation de la première tarde trop. D'autre part, ce dispositif n'évolue pas facilement pour le développeur qui souhaiterait mettre en place des exceptions applicatives.

2. Les exceptions plus sûres que les erreurs

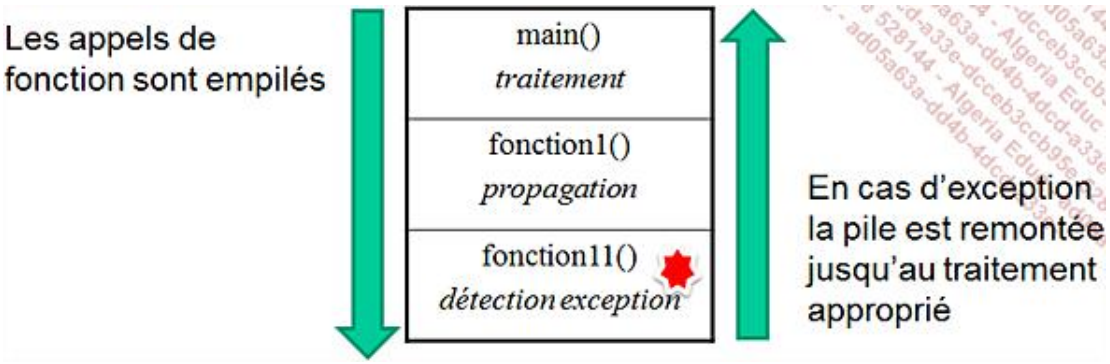
L'énorme avantage des exceptions vient du fait qu'elles sont intrinsèques au langage, et même à l'environnement d'exécution (runtime voire framework). Elles sont donc beaucoup plus sûres pour contrôler l'exécution d'un programme multithread (ce qui est le cas de nombreuses librairies). De plus les exceptions guident le programmeur et lui facilitent la tâche de structuration du code. En d'autres termes il est difficile de s'en passer lorsque l'on maîtrise leur utilisation.

Certains langages vont même plus loin et exigent leur prise en compte à la rédaction du programme. C'est notamment le cas de Java.

Le langage C++ propose des exceptions structurées, bien entendu à base de classes et d'une gestion avancée de la pile. L'idée est de mettre une séquence d'instructions - pouvant contenir des appels à des fonctions - sous surveillance. Lorsqu'un problème survient, le programmeur peut intercepter l'exception décrivant le problème au fil de sa propagation dans la pile.

Le principe des exceptions C++ est de séparer la détection d'un problème de son traitement. En effet, une fonction de calcul n'a sans doute pas les moyens de décider de la stratégie à adopter en cas de défaillance. Les différentes alternatives sont de continuer avec un résultat faux, d'intégrer de nouvelles valeurs saisies par l'utilisateur, de suspendre le calcul...

Lorsqu'une fonction déclenche une exception, celle-ci est propagée à travers la pile des appels jusqu'à ce qu'elle soit interceptée.



Cette organisation réduit à néant les contraintes imposées par la bibliothèque du langage C ; il n'y a plus d'accès

concurrentiel à une variable globale chargée de décrire l'état d'erreur, puisque le contexte de l'erreur est défini par une instance à part entière. Le déclenchement d'une exception provoque un retour immédiat de la procédure ou de la fonction et débute la recherche d'un bloc de traitement.

3. Propagation explicite

Le mot clé **throw** sert à déclencher une exception. Il est suivi d'une instance dont la sémantique est laissée à la charge du programmeur. Parfois le type même utilisé pour instancier l'exception suffit à préciser les circonstances de l'erreur.

Lorsqu'une exception est levée, la pile d'appels est parcourue à la recherche d'un bloc d'interception correspondant à ce type d'exception.

Reprenons notre exemple d'ouverture de fichier et améliorons-le en nous aidant des exceptions :

```
FILE*ouvrir(char*nom)
{
FILE*f;
  f = fopen(nom,"rb");
  if(f==NULL)
    throw 1 ;

  return f;
}

int main(int argc, char* argv[])
{
  FILE*fichier;
  try
  {
    fichier = ouvrir(argv[1]);

    char temp[50];
    fread(temp, 50, sizeof(char),fichier);

    fclose(fichier);
  }
  catch(int code)
  {
    printf("Une erreur est survenue, code=%d\n", code);
  }
  return 0;
}
```

C'est la fonction **ouvrir()** qui décide de déclencher une exception de "valeur" 1 lorsque l'appel à **fopen** échoue. Si tel est le cas, l'instruction **return** n'est pas exécutée, le **throw** interrompant la séquence d'exécution.

Dans la fonction **main()**, les instructions **ouvrir()...fclose()** sont mises sous surveillance au moyen d'un bloc **try** (essai). Si l'une de ces instructions déclenche, même indirectement, une exception, le bloc **try** s'interrompt et l'environnement d'exécution cherche un bloc **catch** (attraper) qui intercepte une exception du type correspondant à celui levé.

4. Types d'exceptions personnalisés

a. Définition de classes d'exception

Notre programme précédent peut facilement être amélioré en créant une classe d'exception. Il est vrai qu'un code entier n'est pas très significatif, alors qu'un type à part entière a tout pour l'être.

```
class OuvertureFichierException
{} ;
```

Nous modifions la séquence de levée d'exception :

```
if(f==NULL)
  throw OuvertureFichierException();
```

En conséquence, le bloc **catch** doit lui-même être aménagé :

```
catch(OuvertureFichierException)
{
  printf("Le fichier n'a pas pu être ouvert\n");
}
```

Maintenant que nous savons créer des types d'exception personnalisés, nous pouvons sans peine prévoir plusieurs blocs catch pour un seul bloc **try**.

```
try
{
  fichier=ouvrir("test.txt");
  char temp[50];
  fread(temp,50,sizeof(char),fichier);
  fclose(fichier);
}
catch(OuvertureFichierException)
{
  printf("Le fichier n'a pas pu être ouvert\n");
}
catch(FermetureFichierException)
{
  printf("Le fichier n'a pas pu être ouvert\n");
}
```

b. Instanciation de classes

Les classes d'exception peuvent être instanciées avec des paramètres qui décrivent le plus finement possible l'erreur. Ainsi, nous pouvons modifier notre classe d'erreur en la dotant d'un constructeur :

```
class OuvertureFichierException
{
public:
  char*message;

  OuvertureFichierException()
  {
    message="";
  }

  OuvertureFichierException(char*msg) : message(msg) {}

  OuvertureFichierException(std::string m)
  {
    using namespace std;
    message=new char[m.length()+1];
    m.copy(message,m.length());
    message[m.length()]=0;
  }

} ;
```

La syntaxe std::string précise que le type string est défini dans l'espace de noms std (espace de noms de la bibliothèque standard STL).

Le code levant l'exception peut maintenant préciser les circonstances de l'erreur :

```
if(f==NULL)
    throw OuvertureFichierException(
std::string("Impossible d'ouvrir ") + nom);
```

En conséquence, nous donnons un nom à l'instance de **OuvertureFichier Exception** pour pouvoir afficher (ou traiter) le maximum d'informations :

```
catch(OuvertureFichierException e)
{
  printf("Le fichier n'a pas pu être ouvert\n");
  printf(e.message);
}
```

c. Classes d'exception dérivées

Il est fréquent de regrouper les classes d'exception ayant trait à la même sémantique d'erreur par le biais d'une dérivation. Dans l'exemple ci-dessous, l'opérateur : indique que les classes **OuvertureFichierException**, **ES_FichierException** et **FermeturefichierException** héritent de la classe **FichierException** (cf. chapitre Programmation orientée objet).

```
class FichierException {} ;

class OuvertureFichierException : public FichierException
{} ;

class ES_FichierException : public FichierException
{} ;

class FermetureFichierException : public FichierException
{} ;
```

L'écriture des blocs **catch** peut pleinement profiter de cette organisation pour filtrer d'abord finement puis grossièrement les causes d'échec :

```
catch(OuvertureFichierException e)
{
    printf("Le fichier n'a pas pu être ouvert\n");
    printf(e.message);
}
catch(FichierException)
{
    printf("Une erreur de type fichier est survenue");
}
```

On doit alors veiller à placer en dernier la classe de base : elle intercepte toutes les exceptions de type **FichierException** qui ne correspondent pas à des cas plus précis, comme **OuvertureFichierException**.

La bibliothèque standard possède un certain nombre d'exceptions dérivées en plusieurs sémantiques : exception générale, exception d'entrée-sortie, exception mathématique...

5. Prise en charge d'une exception et relance

Puisque notre fonction **ouvrir()** est susceptible de déclencher des exceptions, il est important de prévenir le programme qu'un bloc try/catch serait bienvenu. Le mot clé **throw** sert également dans cette situation :

```
FILE*ouvrir(char*nom) throw (OuvertureFichierException)
```

Suivant les compilateurs, l'appel de cette fonction sans bloc **try/catch** lèvera un avertissement, voire une erreur de compilation.

Une fonction qui est marquée **throw (TypeException)** peut se dispenser de la mise en place d'un bloc **try/catch** pour appeler des fonctions marquées avec le même niveau d'exception. Ce principe est d'ailleurs particulièrement utile en Java, langage qui reprend le mécanisme des exceptions de C++ en renforçant le contrôle des marquages.

Si un bloc catch veut relancer une exception qui aurait été déjà interceptée, il peut utiliser le mot clé **throw** seul :

```
catch(TypException)
{
  throw;  // relance
}
```

6. Exceptions non interceptées

La syntaxe **catch(...)** est utile pour intercepter tout type d'exception. Toutefois, l'emploi d'un bloc **try/catch** au niveau de la fonction **main()** peut se révéler contraignant. Certaines exceptions peuvent donc être déclenchées mais non interceptées.

En principe, une telle situation conduit à l'arrêt du programme. Il est également possible d'utiliser la fonction de la bibliothèque standard **std:terminate()** ou bien d'enregistrer une fonction callback à l'aide de la fonction **std:set_terminate()** pour prévenir cette interruption inopportune.

7. Acquisition de ressources

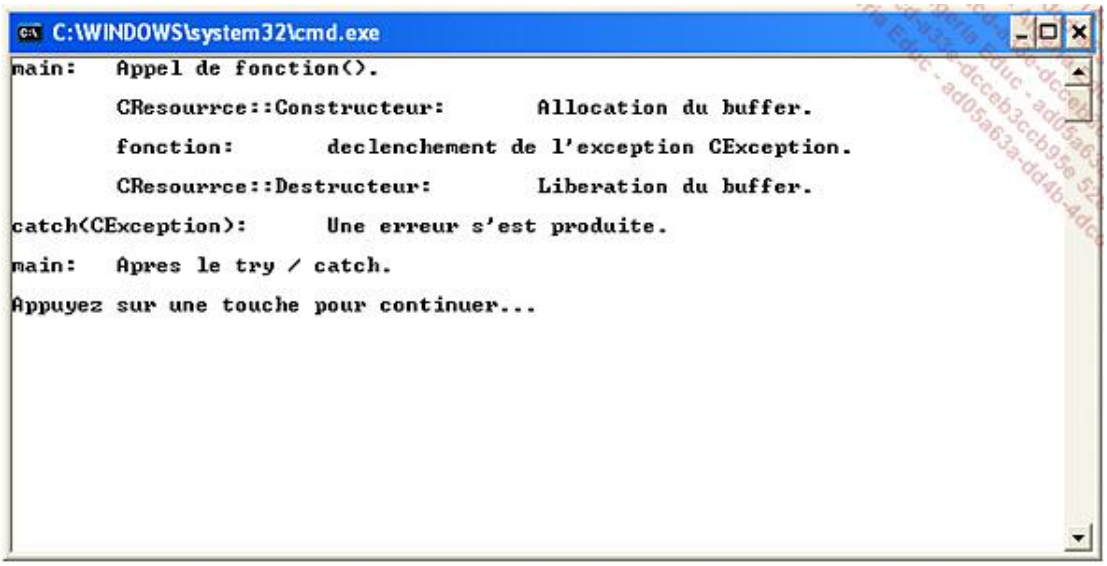
RAII est l'acronyme de *Resource Acquisition Is Initialisation*. Cela signifie que les ressources sont acquises et initialisées au cours d'une opération atomique (insécable). Quel est le rapport avec la gestion des exceptions ? Dans le cas où des objets initialisés depuis une fonction où survient une exception auraient acquis des ressources, il conviendrait de les relâcher avant que le flot d'exécution ne soit dérouté sur le gestionnaire adéquat. Cela tombe bien car C++ libère toujours les objets locaux avant de quitter la portée d'une fonction, même si une exception est levée ou propagée.

L'exemple suit :

```
#include <iostream>
using namespace std;

// affiche un message tabulé
void display_log(char*fname,char*message,int tab)
{
    while(tab-->0) // décompte le nombre de tabulations
        cout << "\t";

    cout << fname << ":\t" << message << endl << endl;
}

// classe d'exception personnalisée
class Cexception
{
public:
    Cexception(){};
```

```
    ~Cexception(){};

    // circonstance de l'exception
    const char *Message() const
    {
        return "Une erreur s'est produite.";
    }

};

const int BUFFER_SIZE = 500;

// type d'objet utilisant des ressources
class Cresource
{
private:
    char*buffer;

public:
    Cresource();
    ~Cresource();
};

Cresource::CResource()
{
    display_log("CResource::Constructeur","Allocation du buffer.",1);
    buffer = new char[BUFFER_SIZE];
}

Cresource::~CResource()
{
    display_log("CResource::Destructeur","Liberation du buffer.",1);
    delete buffer;
}

// fonction instanciant une classe Cresource.
// les objets correspondant allouent 500 octet de mémoire.
void fonction()
{
    CResource resource;
    display_log("fonction","declenchement de l'exception Cexception.",1);

    throw Cexception();
}

// fonction principale
int main()
{
    try
    {
        display_log("main","Appel de fonction().",0);
        fonction();
    }
    catch( CException ex )
    {
        display_log("catch(CException)",(char*)ex.Message(),0);
    }

    display_log("main","Apres le try / catch.",0);
    return 0;
}
```

La trace écran indique bien que l'objet resource est détruit (donc ses ressources internes relâchées) avant que le catch ne capte le flot d'exécution :



Le mécanisme RAII est donc assez proche de la construction try / finally de C# et de Java.

Programmation structurée

Les langages de programmation ont commencé très tôt à assembler les instructions sous la forme de groupes réutilisables, les fonctions. Les variables ont naturellement pris le même chemin, bien qu'un peu plus tardivement.

Le tableau permet de traiter certains algorithmes, à condition que la donnée à traiter soit d'un type uniforme (char, int...). Lorsque la donnée à traiter contient des informations de natures différentes, il faut recourir à plusieurs tableaux, ou bien à un seul tableau en utilisant un type fourre-tout **`void*`**. Il faut bien le reconnaître, cette solution est à proscrire.

À la place, nous définissons des structures regroupant plusieurs variables appelées champs. Ces variables existent en autant d'exemplaires que souhaité, chaque exemplaire prenant le nom d'instance.

Le langage C++ connaît plusieurs formes composites :

- les structures et les unions, aménagées à partir du C ;
- les classes, qui seront traitées au chapitre suivant.

1. Structures

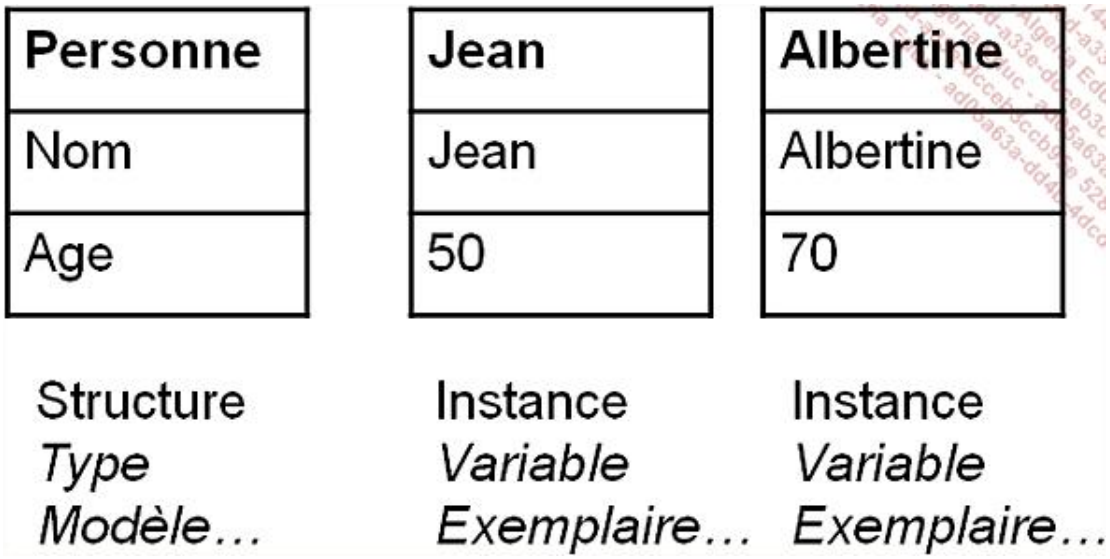
Les structures du C++ - comme celles du C - définissent de nouveaux types de données. Le nom donné à la structure engendre un type de données :

```
struct Personne
{
  char nom[50];
  int age;
} ;
```

À partir de cette structure **`Personne`**, nous allons maintenant créer des variables, en suivant la syntaxe habituelle de déclaration qui associe un type et un nom :

```
Personne jean, albertine;
```

Jean et albertine sont deux variables du type Personne. Comme il s'agit d'un type non primitif - **`char`**, **`int`**..., on dit qu'il s'agit d'instances de la structure Personne. Le terme instance rappelle que le nom et l'âge sont des caractéristiques propres à chaque personne.



On utilise une notation particulière pour atteindre les champs d'une instance :

```
jean.age = 50;  // l'âge de jean
printf("%s",albertine.nom); // le nom d'albertine
```

Cette notation relie le champ à son instance.

a. Constitution d'une structure

Une structure peut contenir un nombre illimité de champ. Pour le lecteur qui découvre ce type de programmation et qui est habitué aux bases de données, il est utile de comparer une structure avec une table dans une base.

C++	SQL
structure	table
champ	champ / colonne
instance	enregistrement

Chaque champ de la structure est bien entendu d'un type particulier. Il peut être d'un type primitif (**char**, **int**...) ou bien d'un type structure. On peut aussi obtenir des constructions intéressantes, par composition :

```
struct Adresse
{
  char *adresse1, *adresse2;
  int code_postal;
  char* ville;
} ;

struct Client
{
  char*nom;
  Adresse adresse;
} ;
```

On utilise l'opérateur point comme aiguilleur pour atteindre le champ désiré :

```
Client cli;
cli.nom = "Les minoteries réunies";
cli.adresse.ville = "Pau";
```

Enfin, il est possible de définir des structures auto-référentes, c'est-à-dire des structures dont un des champs est un pointeur vers une instance de la même structure. Cela permet de construire des structures dynamiques, telles que les listes, les arbres et les graphes :

```
struct Liste
{
  char* element;
  Liste* suite;
} ;
```

Le compilateur n'a aucun mal à envisager cette construction : il connaît fort bien la taille d'un pointeur, généralement 4 octets, donc pour lui la taille de la structure est parfaitement calculable.

b. Instanciation de structures

Il existe plusieurs moyens d'instancier une structure. Nous l'avons vu, une structure engendre un nouveau type de données, donc la syntaxe classique pour déclarer des variables fonctionne très bien :

```
Personne mireille;
```

Instanciation à la définition

La syntaxe de déclaration d'une structure nous réserve une surprise : il est possible de définir des instances aussitôt après la déclaration du type :

```
struct Personne
{
  char*nom;
  int age;
} jean,albertine ;
```

Dans notre cas, jean et albertine sont deux instances de la structure Personne. Bien sûr, il est possible par la suite d'utiliser d'autres modes d'instanciation.

Instanciation par réservation de mémoire

Finalement, l'instanciation agit comme l'allocation d'un espace segmenté pour ranger les champs du nouvel exemplaire. La fonction **`malloc()`** qui alloue des octets accomplit justement cette tâche :

```
Personne* serge = (Personne*) malloc(sizeof(Personne))
```

La fonction **`malloc()`** retournant un pointeur sur **`void`**, on opère un transtypage (**`cast`**) vers le type **`Personne*`** dans le but d'accorder chaque côté de l'égalité. L'opérateur **`sizeof()`** détermine la taille de la structure, en octets.

On accède alors aux champs par l'opérateur **`->`** qui remplace le point :

```
serge->age = 37;
```

Pour réserver plusieurs instances consécutives - un tableau - il faut multiplier la taille de la structure par le nombre d'éléments à réserver :

```
Personne* personnel = (Personne*) malloc(sizeof(Personne)*5)
```

Pour accéder à un champ d'une instance, on combine la notation précédente avec celle employée pour les tableaux :

```
personnel[0]->nom = "Georgette";
personnel[0]->age = 41;
personnel[1]->nom = "Amandine";
personnel[1]->age = 27;
```

La fonction **`malloc()`** est déclarée dans l'en-tête **`<memory.h>`** qu'il faut inclure si besoin. Ce type d'instanciation fonctionnait déjà en langage C, mais l'opérateur **`new`**, introduit en C++, va plus loin.

c. Instanciation avec l'opérateur new

En effet, l'opérateur **`new`** simplifie la syntaxe, car il renvoie un pointeur correspondant au type alloué :

```
Personne*josette = new Personne;
```

Aucun transtypage n'est nécessaire, l'opérateur new appliqué à la structure Personne renvoyant un pointeur (une adresse) de type **Personne***.

Pour réserver un tableau, il suffit d'ajouter le nombre d'éléments entre crochets :

```
Personne*employes = new Personne[10];
```

Là encore la syntaxe est simplifiée, puisqu'il est inutile de préciser la taille de chaque instance. L'opérateur **new** la prend directement en compte, sachant qu'il est appliqué à un type particulier, en l'occurrence **Personne**.

La simplification de l'écriture n'est pas la seule avancée de l'opérateur **new**. Si la structure dispose d'un constructeur (voir à ce sujet le chapitre sur les classes), celui-ci est appelé lorsque la structure est instanciée par le biais de l'opérateur **new**, alors que la fonction **malloc()** se contente de réserver de la mémoire. Le rôle d'un constructeur, fonction "interne" à la structure, est d'initialiser les champs de la nouvelle instance. Nous reviendrons en détail sur son fonctionnement.

d. Pointeurs et structures

Quelle que soit la façon dont a été instanciée la structure, par **malloc()** ou par **new**, l'accès au champ se fait à l'aide de la notation flèche plutôt que point, cette dernière étant réservée pour l'accès à une instance par valeur.

L'opérateur **&** appliqué à une instance a le même sens que pour n'importe quelle variable, à savoir son adresse.

Si cet opérateur est combiné à l'accès à un champ, on peut obtenir l'adresse de ce champ pour une instance en particulier. Le tableau ci-après résume ces modalités d'accès. Pour le lire, nous considérons les lignes suivantes :

```
Personne jean;
Personne* daniel = new Personne;
Personne* personnel = new Personne[10];
```

jean.age	Le champ **age** se rapportant à jean.
daniel->age	Le champ **age** se rapportant à **daniel**, ce dernier étant un pointeur.
&jean	L'adresse de jean. Permet d'écrire **Personne*jean_prime=&jean.**
&jean.age	L'adresse du champ **age** pour l'instance jean.
&daniel	L'adresse du pointeur **daniel**, qui n'est pas celle de l'instance.
&daniel->age	L'adresse du champ **age** pour l'instance **daniel**.
personnel[2]->age	L'âge de la personne portant le numéro 2.
personne[2]	L'adresse de la personne portant le numéro 2.
&personne[2]->age	L'adresse du champ **age** pour la personne portant le numéro 2.

Nous constatons qu'aucune nouveauté n'a fait son apparition. Les notations demeurent cohérentes.

e. Organisation de la programmation

Lorsqu'une structure est créée, il n'est pas rare de la voir définie dans un fichier d'en-tête .h portant son nom. Par la suite, tous les modules **cpp** contenant des fonctions qui vont utiliser cette structure devront inclure le fichier par l'intermédiaire de la directive **#include**.

Inclusion à l'aide de #include

Prenons le cas de notre structure Personne, elle sera définie dans le fichier **personne.h** :

```
// Fichier : personne.h
struct Personne
{
  char nom[50];
  int age;
} ;
```

Chaque module d'extension **.cpp** l'utilisant doit lui-même inclure ce fichier, sachant qu'il est compilé séparément des autres :

```
#include "personne.h"

int main()
{
Personne jean;
  Jean.age=30;
}
```

Protection contre les inclusions multiples

Le système des en-têtes donne de bons résultats mais parfois certains fichiers .h sont inclus plusieurs fois, ce qui conduit à des déclarations multiples du type **Personne**, fait très peu apprécié par le compilateur.

Nous disposons de deux moyens pour régler cette difficulté. Tout d'abord, il est possible d'employer une directive de compilation **#ifndef** suivie d'un **#define** :

```
#ifndef _Personne
#define _Personne

// Fichier : personne.h
struct Personne
{
  char nom[50];
  int age;
} ;
#endif
```

La seconde technique, plus simple, consiste à utiliser une directive propre à un compilateur, **#pragma once**. Cette directive, placée en début de fichier d'en-tête, assure que le contenu ne sera pas accidentellement inclus deux fois. Si le cas se produit, le compilateur recevra une version ne contenant pas deux fois la même définition, donc, nous n'aurons pas d'erreur.

La première technique semble peut-être moins directe, pourtant elle est davantage portable, puisque les directives **#pragma** (pour pragmatique) dépendent de chaque compilateur. Vous êtes bien entendu susceptible de rencontrer les deux dans un programme C++ tiers.

2. Unions

Une union est une structure spéciale à l'intérieur de laquelle les champs se recouvrent. Cette construction particulière autorise un tassement des données, une économie substantielle. Lorsqu'un champ est écrit pour une instance, il écrase les autres puisque tous les champs ont la même adresse. La taille de l'union correspond donc

à la taille du champ le plus large.

Les unions ont deux types d'application. Pour commencer, cela permet de segmenter une structure de différentes manières. Nous pouvons citer comme exemple la structure **address**, qui accueille une union destinée à représenter différents formats d'adresses réseau. En fonction de la nature du réseau - IP, Apple Talk, SPX - les adresses sont représentées de manières différentes.

Comme deuxième type d'application, nous pouvons penser à la représentation des nombres. Suivant que nous souhaitons privilégier la vitesse ou la précision des calculs, nous pouvons utiliser une union pour calculer en int, en float ou en double. Utiliser une structure ne serait pas une bonne idée car chaque "nombre" occuperait 4+4+8 soit 16 octets. L'union donne de meilleurs résultats :

```
union Valeur
{
    int nb_i;
    float nb_f;
    double nb_d;
} ;
```

La taille de cette union égale 8 octets, soit l'espace occupé par un double.

Il n'est pas rare d'inclure une union dans une structure, un champ supplémentaire indiquant lequel des champs de l'union est "actif" :

```
enum TNB { t_aucun,t_int, t_float, t_double };

struct Nombre
{
  char t_nb;
  union Valeur
  {
    int nb_i;
    float nb_f;
    double nb_d;
  } val ;
} ;
```

Nous pouvons à présent imaginer quelques fonctions pour travailler avec cette construction :

```
void afficher(Nombre& n)
{
  switch(n.t_nb)
  {
  case t_int:
    printf("%d\t",n.val.nb_i);
    break;
  case t_float:
    printf("%f\t",n.val.nb_f);
    break;
  case t_double:
    printf("%f\t",n.val.nb_d);
    break;
  }
}

Nombre* lire_int()
{
  Nombre* c=new Nombre;
  c->t_nb=t_int;
  printf("entier: ");
  scanf("%d",&c->val.nb_i);

  return c;
}
```

```
Nombre* lire_float()
{
  Nombre* c=new Nombre;
  c->t_nb=t_float;
  printf("décimal: ");
  scanf("%f",&c->val.nb_f);
  //cin >> c->val.nb_f; // cin >> float&

  return c;
}
Nombre lire_double()
{
  Nombre c;
  c.t_nb=t_double;
  printf("double: ");
  scanf("%lg",&c.val.nb_d);
  return c;
}
```

Dans cet exemple, nous avons mélangé différents modes de passage ainsi que différents modes de retour (valeur, adresse...).

Pour ce qui est de l'instanciation et de l'accès aux champs, l'union adopte les mêmes usages (notations) que la structure.

3. Copie de structures

Que se passe-t-il lorsque nous copions une structure par valeur dans une autre structure ? Par exemple, que donne l'exécution du programme suivant ?

```
struct Rib
{
  char*banque;
  int guichet;
  int compte;
} ;

int main(int argc, char* argv[])
{
  Rib compte1,compte2;
  compte1.banque = "Banca";
  compte1.guichet = 1234;
  compte1.compte = 555666777;

  compte2 = compte1;

  printf("compte1, %s %d %d\n",compte1.banque,compte1.guichet,
compte1.compte);
  printf("compte2, %s %d %d\n",compte2.banque,compte2.guichet,
compte2.compte);
  return 0;
}
```

L'exécution de ce programme indique que les valeurs de chaque champ sont effectivement copiées de la structure **compte1** à la structure **compte2**. Il faut cependant faire attention au champ **banque**. S'agissant d'un pointeur, la modification de la zone pointée affectera les deux instances :

```
Rib compte1,compte2;

  // alloue une seule chaîne de caractères
  char* banque=new char[50];
  strcpy(banque,"Banque A");
```

```
  // renseigne la première instance
  compte1.banque = banque;
  compte1.guichet = 1234;
  compte1.compte = 555666777;

  // copie les valeurs dans la seconde
  compte2 = compte1;

  // affichage
  printf("compte1, %s %d %d\n",compte1.banque,compte1.guichet,
compte1.compte);
  printf("compte2, %s %d %d\n",compte2.banque,compte2.guichet,
compte2.compte);

  // apparemment, on modifie uniquement compte1
printf("\nModification de compte1\n");
  strcpy(compte1.banque,"Banque B");

  // vérification faite, les deux instances sont sur "Banque B"
  printf("compte1, %s %d %d\n",compte1.banque,compte1.guichet,
compte1.compte);
  printf("compte2, %s %d %d\n",compte2.banque,compte2.guichet,
compte2.compte);
```

Vérification faite, l'exécution indique bien que les deux pointeurs désignent la même adresse :

La copie des structures contenant des champs de type pointeur n'est pas la seule situation à poser des problèmes. Considérons à présent l'extrait suivant :

```
Rib *c1,*c2;
 c1 = &compte1;
 c2 = c1;
 c1->compte = 333;
 printf("c2, %s %d %d\n",c2->banque,c2->guichet,c2->compte);
```

L'exécution indique 333 comme numéro de compte pour **c2**. Autrement dit, **c1** et **c2** désignent le même objet, et l'affectation **c2=c1** n'a rien copié du tout, sauf l'adresse de l'instance.

Il aurait été plus judicieux d'utiliser la fonction **memcpy()** :

```
c2 = new Rib;
 memcpy(c2,c1,sizeof(Rib));
 c1->compte = 222;
 printf("c2, %s %d %d\n",c2->banque,c2->guichet,c2->compte);
```

Cette fois, l'exécution indique bien que **c2** et **c1** sont indépendants. C'est l'allocation par **new** (ou **malloc()**) qui aura fait la différence.

4. Création d'alias de types de structure

L'instruction **typedef** étudiée au chapitre précédent sert également à définir des alias (de types) de structure :

```
// une structure décrivant un nombre complexe
struct NombreComplexe
{
    double reel,imaginaire;
};

typedef NombreComplexe Complexe;    // alias de type
typedef NombreComplexe* Pcomplexe;    // alias de type pointeur
typedef NombreComplexe& Rcomplexe;    // alais de type référence

int main()
{
    Complexe c1;
    PComplexe pc1 = new Complexe;
    RComplexe rc1 = c1;

    return 0;
}
```

Cet usage est généralement dévolu aux API systèmes qui définissent des types puis des appellations pour différents environnements. Par exemple le **`char*`** devient **LPCTSTR** (long pointer to constant string), le pointeur de structure **`Rect`** devient **LPRECT**…

5. Structure et fonction

Ce sont les fonctions qui opèrent sur les structures. Il est fréquent de déclarer les structures comme variables locales d'une fonction dans le but de les utiliser comme paramètres d'autres fonctions. Quel est alors le meilleur moyen pour les transmettre ? Nous avons à notre disposition les trois modes habituels, par valeur, par adresse (pointeur) ou par référence.

a. Passer une structure par valeur comme paramètre

Le mode par valeur est indiqué si la structure est de petite taille et si ses valeurs doivent être protégées contre toute modification intempestive de la part de la fonction appelée. Ce mode implique la recopie de tous les champs de l'instance dans la pile, ce qui peut prendre un certain temps et consommer des ressources mémoire forcément limitées. Dans le cas des fonctions récursives, la taille de la pile a déjà tendance à grandir rapidement, il n'est donc pas judicieux de la surcharger inutilement.

Toutefois, cette copie empêche des effets de bord puisque c'est une copie de la structure qui est passée.

```
void afficher(Nombre n)
{
  switch(n.t_nb)
  {
  case t_int:
    printf("%d\t",n.val.nb_i);
    break;
  case t_float:
    printf("%f\t",n.val.nb_f);
    break;
  case t_double:
    printf("%f\t",n.val.nb_d);
    break;
  }
}
```

b. Passer une structure par référence comme paramètre

Ce mode constitue une avancée considérable, puisque c'est la référence (adresse inaltérable) de la structure qui est transmise. Cette information occupe 4 octets (en compilation 32 bits) et autorise les effets de bord sous

certaines conditions. De plus, le passage par référence est transparent pour le programmeur, ce dernier n'ayant aucune chance de passer une valeur littérale comme instance de structure.

```
void afficher(Nombre& n)
```

Nous verrons comment la structure (classe) peut être aménagée de manière à ce que l'accès en modification des champs puisse être contrôlé de manière fine.

Il est possible de protéger la structure en déclarant le paramètre à l'aide du mot clé **const** :

```
void afficher(const Nombre& n)
{
  n.val.nb_i=10;  // erreur, n invariable
}
```

c. Passer une structure par adresse comme paramètre

Ce mode continue à être le plus utilisé, sans doute car il est la seule alternative au passage par valeur autorisée en langage C. Il nécessite parfois d'employer l'opérateur **&** à l'appel de la fonction, et le pointeur reçu par la fonction se conforme à l'arithmétique des pointeurs. Enfin, ce mode autorise les effets de bord.

```
void afficher(Nombre* n)
```

d. De la programmation fonctionnelle à la programmation objet

En admettant que certaines fonctions puissent migrer à l'intérieur de la structure, dans le but évident de s'appliquer aux champs d'une instance en particulier, nous découvrons cette notion de la programmation orientée objet que l'on appelle l'encapsulation, c'est-à-dire la réunion d'une structure et de fonctions. Cette construction est tout à fait légale en C++ :

```
struct Nombre
{
  char t_nb;
  union Valeur
  {
    int nb_i;
    float nb_f;
    double nb_d;
  } val ;

  void afficher()
  {
    switch(t_nb)
    {
      case t_int:
          printf("%d\t",val.nb_i);
          break;
      case t_float:
          printf("%f\t",val.nb_f);
          break;
      case t_double:
          printf("%f\t",val.nb_d);
          break;
    }
  }
} ;
```

Connaissant un **nombre**, il est très facile de modifier les notations pour utiliser cette méthode **afficher()** en remplacement de la fonction **afficher(Nombre&)** :

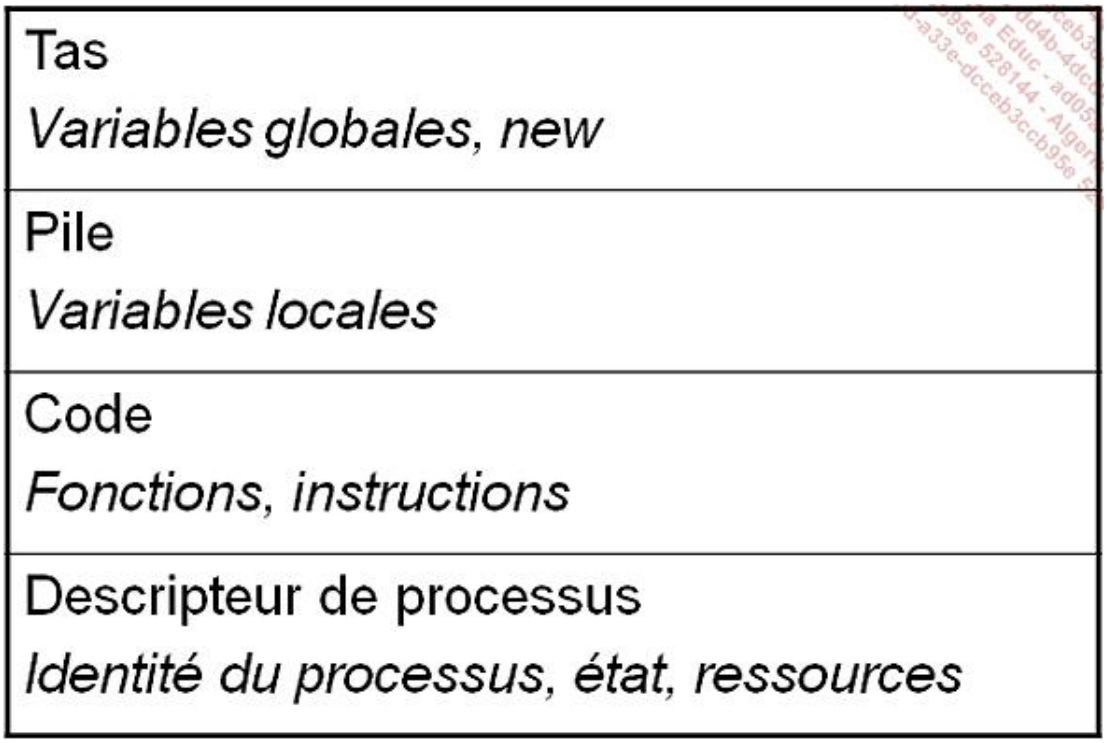
```
Nombre a;
   a =lire_float();  // utilise la fonction lire_float()
  a.afficher();  // utilise la méthode afficher()
  afficher(a);  // utilise la fonction afficher()
```

Nous verrons au chapitre suivant ce qui distingue la programmation fonctionnelle et la programmation orientée objet. Pour le lecteur qui connaît la notion de visibilité dans une classe, il est utile de préciser que les membres d'une structure (champs ou méthodes) sont publics par défaut.

Gestion de la mémoire

La mémoire est vitale pour le système d'exploitation dont la tâche principale consiste à séparer chaque processus des autres, en lui allouant une quantité initiale de mémoire. Nous découvrons différentes stratégies d'allocation en fonction des systèmes. Les plus anciens (MS-DOS) et les plus basiques (sur microcontrôleur) adoptent une allocation par blocs d'une taille déterminée (64 ko par exemple), cette taille n'étant pas amenée à évoluer au cours du temps. D'autres systèmes ont recours à la pagination - une segmentation fine de l'espace mémoire en blocs de 4 ko - ce qui autorise une gestion dynamique et efficace de la mémoire. Consécutivement à l'emploi de la mémoire paginée, l'adresse exprimée par un pointeur n'est jamais une adresse physique, mais logique, le microprocesseur se chargeant de la traduction.

Pour ce qui est du modèle applicatif, c'est-à-dire de la segmentation de la mémoire du processus, nous avons en général les zones suivantes :



Le langage C++ étant comme son prédécesseur très proche du système d'exploitation, il convient de bien connaître les différents types de gestion de la mémoire.

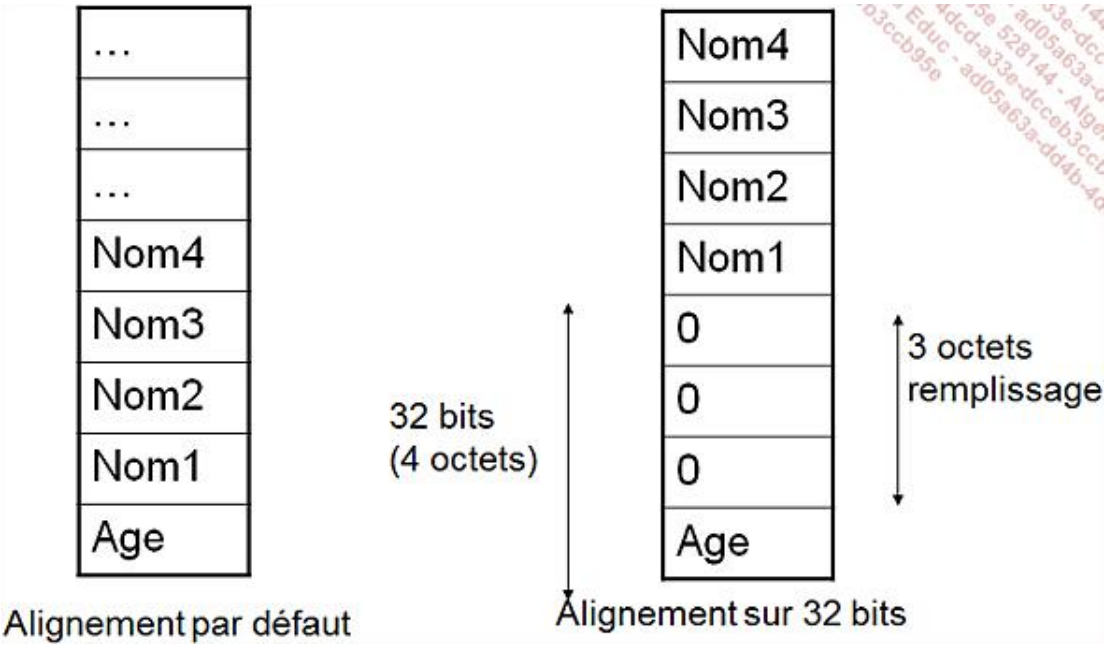
1. Alignement des données

Le développeur doit bien prendre en compte cet aspect de la gestion de la mémoire si la portabilité est un critère important. Tous les compilateurs et tous les microprocesseurs ne rangent pas les données de la même façon.

Prenons le cas de la structure **Personne** :

```
struct Personne
{
  char age;
  char*nom;
} ;
```

Sachant que le microprocesseur Pentium lit la mémoire par blocs de 4 octets, nombre de compilateurs insèreront 3 octets de remplissage pour aligner le champ nom sur le prochain bloc. Il en résulte une taille de structure égale à 8 alors que le comptage manuel donne 5.



Aussi, beaucoup de microprocesseurs utilisent la représentation big-endian, ce qui fait que les données sont rangées en mémoire en commençant par le poids fort. Le Pentium utilise la convention inverse. Ces caractéristiques sont déterminantes lorsqu'un fichier est échangé entre deux plates-formes, même s'il s'agit du même programme C++, compilé pour chaque plate-forme, soit avec deux compilateurs différents.

2. Allocation de mémoire interprocessus

Nous connaissons déjà deux moyens d'allouer de la mémoire dynamiquement. La fonction **`malloc()`** et l'opérateur **`new`** ont une vocation "langage". Dans certaines situations, il est nécessaire de faire appel à d'autres fonctions offertes par le système. La mémoire partagée, utilisée par le Presse-papiers de Windows est un bon exemple.

La communication interprocessus est une prérogative du système d'exploitation. Comme beaucoup d'entre eux sont programmés en langage C ou C++, il est assez facile d'utiliser ces fonctions. Toutefois, la notion de pointeur n'existe pas dans tous les langages, et les fonctions d'allocation retournent souvent un entier, appelé **`handle`**, identifiant le bloc alloué. La désignation de cet entier prend souvent comme nom **`Hresult`** (Handle Result), qui est en fait un entier déguisé.

Nous retrouverons des exemples de ce type au chapitre consacré à la programmation C++ sous Windows.

La bibliothèque standard du C

Avec les structures et l'allocation de la mémoire, nous nous rendons compte qu'un langage doit s'appuyer sur des librairies système pour construire des applications complètes. Le langage C++ possède sa propre librairie, mais nombre de programmeurs utilisent toujours les fonctions standards du langage C.

1. Les fonctions communes du langage C <stdlib.h>

La librairie standard **stdlib.h** contient des fonctions d'ordre général. Certaines fonctions peuvent être d'ailleurs déclarées dans d'autres en-têtes.

Voici une liste résumant quelques fonctions intéressantes pour le développement courant. Il est judicieux de consulter l'ouvrage de Kernighan et Ritchie ou bien une documentation fournie avec le compilateur pour connaître à la fois la liste complète des fonctions et en même temps leur signature.

Le Kernighan et Ritchie est l'ouvrage de référence écrit par les créateurs du langage C. Il est toujours édité et mis à jour à partir des évolutions du langage C.

Fonctions	Utilité
atoi, **atof**, **strtod**...	Fonctions de conversion entre un type chaîne et un type numérique.
getenv, **setenv**	Accès aux variables d'environnement système.
malloc, **calloc**	Allocation de mémoire.
rand, **abs**	Fonctions mathématiques.

La bibliothèque standard stdlib contient aussi des macros instructions basées sur la syntaxe **#define** :

Macro	Utilité
__min, **__max**	Donne les valeurs min et max de deux arguments
NULL	Littéralement **(void*)0**

Un petit exemple montre comment utiliser la bibliothèque :

```
char* lecture = new char[500];
printf("Nombre ? ");
scanf("%s",lecture);

double nombre = atof(lecture);
double pi = 3.14159265358;

printf("Le nombre le plus grand est %2f\n",__max(nombre,pi));
```

2. Chaînes <string.h>

Le langage C++ gère toujours ses littérales de chaîne au format **`char*`** défini par le langage C. Aussi est-il important de connaître les principales fonctions de la bibliothèque **`string.h`**.

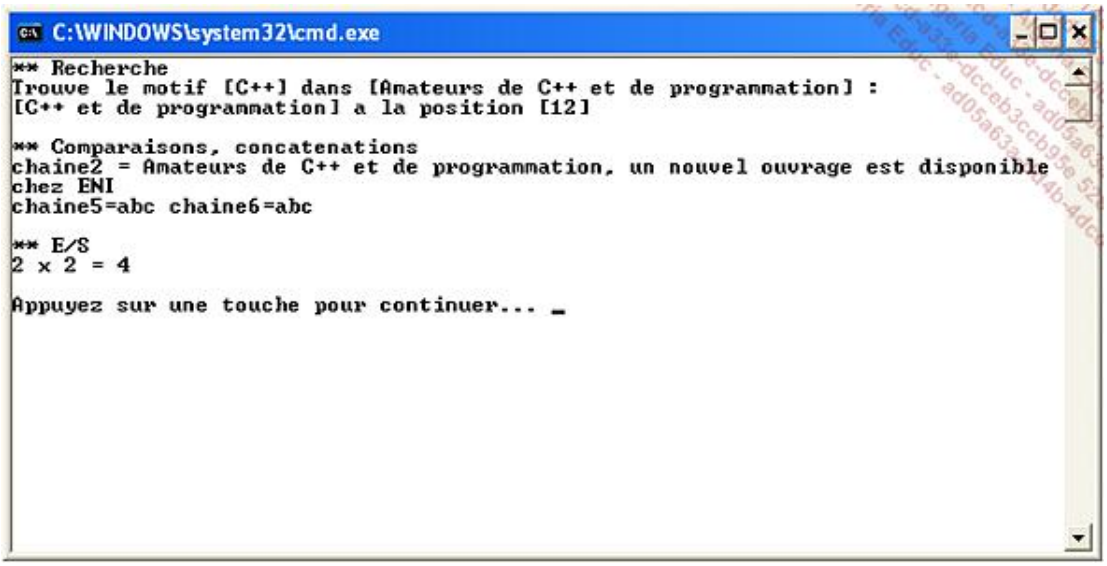
Fonctions	Utilité
`memcpy`, **`memcmp`**, **`memset`**	Vision mémoire des chaînes de caractères : copie, comparaison, initialisation
`strcpy`, **`strcmp`**, **`strcat`**, **`strlen`**	Copie, comparaison, concaténation, longueur
`strchr`, **`strstr`**	Recherche de caractère, de sous-chaîne
`strlwr`, **`strupr`**	Conversion minuscule/majuscule

Prenons là encore un exemple pour illustrer l'emploi de cette librairie :

```
/*
    * Recherche
    */
    printf("** Recherche\n");

    // chaîne à analyser
    char* chaine1 = "Amateurs de C++ et de programmation";

    // motif à rechercher
    char* motif = "C++";

    // recherche : strstr fournit un pointeur sur la sous-chaîne
    char* souschaine = strstr(chaine1,motif);

    // résultat
    if(souschaine != NULL)
        printf("Trouve le motif [%s] dans [%s] :\n[%s] a la position [%d]\n\n",
            motif,chaine1,souschaine,souschaine-chaine1);

    /*
     * Comparaison, concaténation
     */

    printf("** Comparaisons, concatenations\n");
```

```
    // réserver de la mémoire
    char* chaine2 = new char[strlen(chaine1) + 50];

    // construire la chaîne
    strcpy(chaine2,chaine1);
    strcat(chaine2,", un nouvel ouvrage est disponible chez ENI");

    // résultat
    printf("chaine2 = %s\n",chaine2);

    // initialiser une chaîne
    char chaine3[50];
    memset(chaine3,'O',3); // chaque octet est initialisé par le caractère O
    chaine3[3]=0; // 0 terminal (identique à '\0' à la 4ème position

    // résultat
    if(strcmp(chaine3,"OOO"))
        printf("chaine3 = %s\n",chaine3);

    // ne pas se tromper avec les longueurs
    char* chaine4 = "ABC";
    char* chaine5 = new char[ strlen(chaine4) + 1]; // +1 pour le 0 terminal
    strcpy(chaine5, chaine4);

    // passer en minuscule
    char* chaine6 = strlwr(chaine5); // attention chaine5 est modifiée
    printf("chaine5=%s chaine6=%s\n\n", chaine5, chaine6);

    /*
     * E/S
     */

    printf("** E/S\n");

    char message[500];

    // écrire un message formaté dans une chaîne
    sprintf(message,"2 x 2 = %d\n", 2*2);

    printf("%s\n",message);
```

L'exécution de ce programme nous donne le résultat suivant :



3. Fichiers <stdio.h>

Les fonctions déclarées dans **stdio.h** sont consacrées aux opérations d'entrée-sortie, au sens large. On trouve deux types de prise en charge des fichiers : par **handle** système et par structure **FILE***. Le premier type est plus ancien et fortement lié à Unix. Toutefois, la technique du **handle** a été généralisée dans d'autres parties du système d'exploitation : les sockets et le réseau, le graphisme...

Un handle est généralement assimilable à un entier. Il s'agit d'un identificateur géré par le système pour désigner différents types de ressources.

L'accès par la structure **FILE*** est quant à lui de plus haut niveau, donc plus spécialisé. Certains "fichiers" sont déjà déclarés, tels que **stdin**, **stdout**, **stderr** et **stdprn**.

Enfin, **stdio.h** propose des fonctions de formatage vers des chaînes de caractères, telles que **sprintf** (également présente dans **string.h**).

Fonctions	Utilité
printf, **scanf**	Impression sur **stdout**, lecture depuis **stdin**
fprintf, **fscanf**	Impression et lecture depuis un **FILE***
sprintf	Impression sur une chaîne
open, **close** , **read**, **write**	Ouverture, fermeture, lecture, écriture sur un **handle** (entier)
fopen, **fclose**, **fread**, **fwrite**	Ouverture, fermeture, lecture et écriture sur un **FILE***
fflush	Transfert du tampon (buffer) vers son périphérique (**FILE***)
putc, **getc**	Écriture, lecture d'un seul caractère (**handle**)
fputc, **fgetc**	Écriture, lecture d'un seul caractère depuis un **FILE***
feof	Test de la fin d'un flux (**FILE***)
ftell, **fseek**	Lecture et déplacement du pointeur de lecture /écriture dans un fichier **FILE***
remove, **rename**, **unlink**, **tempnam**	Gestion du nom d'un fichier

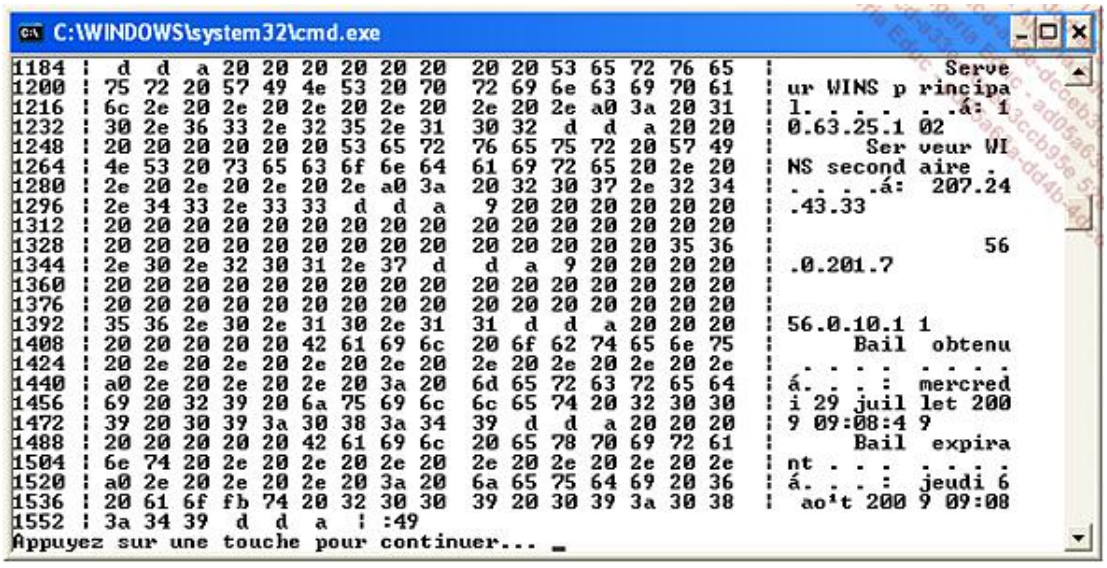
L'exemple est ici un peu plus développé que précédemment. Il s'agit d'afficher le contenu d'un fichier en hexadécimal et en ASCII (on dit dumper). Le programme contient des méthodes utilitaires de déplacement dans le fichier ainsi qu'une fonction de lecture complète du fichier. Le lecteur pourra à titre d'exercice remplacer cette fonction par un menu proposant à l'utilisateur de lire ligne à ligne le fichier, de revenir au début…

```
// définit le point d'entrée pour
l'application console.
//

#include "stdafx.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

const int T_BLOC = 16;

// revient au début du fichier
void debut(FILE* f)
{
    fseek(f,0,SEEK_SET);
```

```
}

// va à la fin du fichier
void fin(FILE* f)
{
    // longueur du fichier
    fseek(f,0,SEEK_END);
    long taille = ftell(f);
    fseek(f,taille - (taille % T_BLOC), SEEK_SET);
}

// remonte dans le fichier
void haut(FILE* f)
{
    fseek(f, -T_BLOC, SEEK_CUR);
}

// descend dans le fichier
void bas(FILE* f)
{
    fseek(f, T_BLOC, SEEK_CUR);
}

// détermine la taille du fichier
long taille(FILE* f)
{
    // position actuelle
    long pos = ftell(f);

    // va à la fin et détermine la longeur à partir de la position
    fseek(f, 0, SEEK_END);
    long taille = ftell(f);

    // revient au point de début
    fseek(f, pos, SEEK_SET);

    return taille;
}

void afficher_bloc(FILE* f)
{
    int offset = ftell(f);
    char chaine_hexa[T_BLOC*3 + 2 + 1];
    char chaine_ascii[T_BLOC*2 + 2 + 1];

    strcpy(chaine_hexa,"");
    strcpy(chaine_ascii,"");

    int data;
    for(int i=0;i<16;i++)
    {
        int car = fgetc(f);
        if(car == -1)
            break;

        char concat[50];

        sprintf(concat,"%2x ",car);
        strcat(chaine_hexa,concat);

        sprintf(concat,"%c",car>=32?car:' ');
        strcat(chaine_ascii,concat);

        if(i == T_BLOC/2)
        {
            strcat(chaine_hexa," ");
            strcat(chaine_ascii," ");
        }
    }
```

```
    // fprintf(stdout) = printf !
    fprintf(stdout,"%4d | %s | %s\n",offset,chaine_hexa,chaine_ascii);
}

int main(int argc, char* argv[])
{

    char* nom_fichier="c:file://temp//briceip.txt";
    FILE*f = NULL;

    try {
        f = fopen(nom_fichier,"rb");

        /*
         * lecture continue du fichier
         */
        long p = 0;
        long t = taille(f);

        debut(f);
        while(p<t)
        {
            afficher_bloc(f);
            p+=T_BLOC;
        }

        fclose(f);
    }
    catch(...)
    {
        printf("erreur\n");
    }


    return 0;
}
```

Voici le résultat pour un fichier d'exemple :



Classes et instances

L'objectif poursuivi par Bjarne Stroustrup était, rappelons-le, d'implémenter sur un compilateur C les classes décrites par le langage Simula. Ces deux derniers langages étant radicalement opposés dans leur approche, il fallait identifier une double continuité, notamment du côté du C.

Il fut aisé de remarquer que la programmation serait grandement simplifiée si certaines fonctions pouvaient migrer à l'intérieur des structures du C. De ce fait, il n'y aurait plus de structure à passer à ces fonctions puisqu'elles s'appliqueraient évidemment aux champs de la structure.

Toutefois, il fallait conserver un moyen de distinguer deux instances et c'est pour cette raison que l'on a modifié la syntaxe de l'opérateur point :

Programmation fonctionnelle	Programmation orientée objet

`struct Point`
`{`
`int x,y ;`
`} ;`
`void afficher(Point p)`
`{`
`printf("%d,%d\n",p.x,p.y);`
`}`
	`struct Point`
	`{`
	`int x,y;`
	`void afficher()`
	`{`
	`printf("%d,%d\n",x,y);`
	`}`
	`} ;`

`Point p1;` `afficher(p1);`	`Point p1;` `p1.afficher();`

Cette différence d'approche a plusieurs conséquences positives pour la programmation. Pour commencer, le programmeur n'a plus à effectuer un choix parmi les différents modes de passage de la structure à la fonction **`afficher()`**. Ensuite, nous allons pouvoir opérer une distinction entre les éléments (champs, fonctions) de premier plan et de second plan. Ceux de premier plan seront visibles, accessibles à l'extérieur de la structure. Les autres seront cachés, inaccessibles.

Ce procédé garantit une grande indépendance dans l'implémentation d'un concept, ce qui induit également une bonne stabilité des développements.

1. Définition de classe

Une classe est donc une structure possédant à la fois des champs et des fonctions. Lorsque les fonctions sont considérées à l'intérieur d'une classe, elles reçoivent le nom de méthodes.

L'ensemble des champs et des méthodes est désigné sous le terme de membres. Nous ne recommandons pas la désignation des champs à l'aide du terme attribut, car il peut prendre un sens très particulier en langage C++ managé ou en langage C#.

Pour le lecteur qui passe de Java à C++, il faut faire attention à terminer la déclaration d'une classe par le caractère point-virgule, une classe étant la continuité du concept de structure :

```
class Point
{
int x,y ;  // deux champs
    // une méthode
    void afficher()
    {
        printf("%d,%d\t",x,y);
```

```
    }
} ; // point-virgule
```

La classe suit globalement les mêmes règles que la structure pour ce qui est de son utilisation : instanciation / allocation, copie, passage à une fonction...

a. Les modificateurs d'accès

Nous allons à présent opérer une distinction entre les méthodes (fonctions) accessibles depuis l'extérieur de la classe et celles qui n'ont qu'une utilité algorithmique. De même, certains champs sont exposés à l'extérieur de la classe, leur accès en lecture et en modification est autorisé, alors que d'autres doivent être protégés contre des accès intempestifs.

Cette distinction laisse une grande latitude dans l'organisation d'une classe, la partie cachée pouvant évoluer sans risque de remettre en question le reste du programme, la partie accessible étant au contraire considérée comme stable.

Le langage C++ offre plusieurs niveaux d'accessibilité et la palette de possibilités est assez large, si bien que certains langages qui lui succèdent dans l'ordre des publications ne les retiennent pas tous.

Modificateur	Accessibilité
public	Complète. Le membre - champ ou méthode - est visible à l'intérieur de la classe comme à l'extérieur.
private	Très restreinte. Le membre n'est accessible qu'à l'intérieur de la classe.
protected	Restreinte à la classe courante et aux classes dérivées.
friend	Restreinte à une liste de fonctions identifiées comme étant amies.

Donnons maintenant un exemple de visibilité. Sachant que la visibilité par défaut dans une classe est **private**, nous allons choisir pour chaque membre un modificateur d'accès :

```
class Point
{
private:
  int x,y;
public:
  void afficher()
  {
    printf("%d,%d\t",x,y);
  }
  void positionner(int X,int Y)
  {
    x=X;
    y=Y;
  }
  int couleur;
} ;
```

Avant d'utiliser cette classe, dressons un tableau des visibilités pour chaque membre de la classe Point :

x	private (caché)
y	private (caché)
afficher	public
positionner	public

couleur	public

Les champs **x** et **y** sont bien présents pour chaque instance, mais ils ne sont lisibles/modifiables que par des méthodes appartenant à la classe, comme **afficher()** et **positionner()**. Les autres fonctions, même si elles appartiennent à des classes, n'ont pas d'accès direct à **x** et **y**. Elles doivent passer par des méthodes publiques de la classe Point.

Le champ **couleur** est lui public, ce qui signifie qu'il est complètement exposé.

Essayons maintenant d'appliquer ces règles de visibilité à la classe sus-décrite.

```
Point p;
  p.x=3;  // erreur, x est privé
  p.y=2; // erreur, y est privé
  p.positionner(89,2);  // ok, positionner() est publique
  p.afficher();  // ok, afficher() est publique
  p.couleur=0x00FF00FF;  // ok, couleur est public
```

Lorsque les commentaires indiquent erreur, cela signifie que le compilateur relèvera la ligne comme n'étant pas valide. Le message circonstanciel peut avoir la forme "champ inaccessible" ou "champ privé", voire "champ invisible". Parfois le compilateur est plus explicite en indiquant que la portée privée (**private**) d'un membre ne convient pas à l'usage que l'on en fait.

Pour découvrir des exemples de champs **protected** ou de fonctions amies, nous devrons attendre encore quelques pages.

Pour le programmeur qui débute, le choix d'une portée n'est pas facile à opérer. Et il faut éviter de suivre le conseil suivant qui se révèle trop stéréotypé, un peu grossier : tous les champs sont privés (**private**) et toutes les méthodes sont publiques. Cette règle peut convenir dans certains cas, mais sans doute pas dans tous les cas de figure. D'abord, elle favorise trop la notion d'interface au détriment de l'algorithmie (la programmation fonctionnelle). Or une classe ne se résume pas à une interface, sans quoi l'encapsulation - la réunion d'une structure et de fonctions - n'aurait aucun intérêt. Les détails d'implémentation ont besoin d'être cachés pour évoluer librement sans remettre en question le reste du programme, fort bien. Ce qui nous conduit à déclarer certaines méthodes avec un niveau de visibilité inférieur à public. De même, certains champs ont un typage particulièrement stable, et si aucun contrôle n'est nécessaire quant à leur affectation, il n'y a aucune raison de les déclarer de manière privée.

Bien que le choix d'une visibilité puisse être décidé par le concepteur qui s'aidera d'un système de modélisation comme UML, le programmeur à qui incomberait cette responsabilité peut s'aider du tableau suivant pour décider quelle visibilité choisir. En l'absence de dérivation (héritage), le nombre de cas de figure est assez limité, et il faut bien reconnaître que les programmes contenant un nombre élevé de dérivations ne sont pas légion, surtout sans l'emploi d'UML.

champ algorithmique	**private** ou **protected**
champ de structure	**public**
champ en lecture seule	**private** ou **protected** avec une méthode **getXXX()** publique
champ en écriture seule	**private** ou **protected** avec une méthode **setXXX()** publique
champ en lecture et écriture avec contrôle des accès	**private** ou **protected**, avec deux méthodes publiques **getXXX()** et **setXXX()**
champ "constante" de classe	champ public, statique et sans doute déclaré **const**
méthode caractérisant les opérations accessibles à un objet	Publique. La méthode fait alors partie de l'interface

Méthode destinée à porter l'algorithmie	**private** ou **protected**

Nous constatons dans ce tableau que les fonctions amies n'y figurent pas. Il s'agit d'un concept propre à C++, donc très peu portable, qui a surtout de l'utilité pour la surcharge des opérateurs lorsque le premier argument n'est pas du type de la classe (reportez-vous à la partie Autres aspects de la POO - Surchages d'opérateurs sur la surcharge pour d'autres détails).

Également, il est fait plusieurs fois mention du terme "interface". Une interface est une classe qui ne peut être instanciée. C'est un concept. Il y a deux façons d'envisager l'interface, selon qu'on la déduit d'une classe ordinaire ou bien que l'on construit une classe ordinaire à partir d'une interface. Dans le premier cas, on déduit l'interface d'une classe en créant une liste constituée des méthodes publiques de la classe. Le concept est bâti à partir de la réalisation concrète. Dans le second cas, on crée une classe dérivant (héritant) de l'interface. Le langage C++ n'a pas de terme spécifique pour désigner les interfaces, bien qu'il connaisse les classes abstraites. On se reportera à la section Héritage - Méthodes virtuelles et méthodes virtuelles pures sur les méthodes virtuelles pures pour terminer l'étude des interfaces.

Quoi qu'il en soit, c'est une habitude en programmation orientée objet (POO) de désigner l'ensemble des méthodes publiques d'une classe sous le terme d'interface.

b. Organisation de la programmation des classes

Il existe avec C++ une organisation particulière de la programmation des classes. Le type est défini dans un fichier d'en-tête .h, comme pour les structures, alors que l'implémentation est généralement déportée dans un fichier source **.cpp**. Nous nous rendrons compte par la suite, en étudiant les modules, que les notations correspondantes restent très cohérentes.

En reprenant la classe Point, nous obtiendrons deux fichiers, **point.h** et **point.cpp**. À la différence du langage Java, le nom de fichier n'a d'importance que dans les inclusions et les makefiles. Comme il est toujours explicité par le programmeur, aucune règle syntaxique n'impose de noms de fichier pour une classe. Toutefois, par souci de rigueur et de cohérence, il est d'usage de nommer fichier d'en-tête et fichier source à partir du nom de la classe, en s'efforçant de ne définir qu'une classe par fichier. Donc si la classe **PointColore** vient compléter notre programme, elle sera déclarée dans **PointColore.h** et définie dans **PointColore.cpp**.

Voici pour commencer la déclaration de la classe **Point** dans le fichier **point.h** :

```
class Point
{
private:
  int x,y;
public:
  void afficher();
  void positionner(int X,int Y);
  int couleur;
} ;
```

Nous remarquons que les méthodes sont uniquement déclarées à l'aide de prototypes (signature close par un point-virgule). Cela suffit aux autres fonctions pour les invoquer, peu importe où elles sont réellement implémentées.

Ensuite, nous implémentons (définissons) la classe **Point** dans le fichier **point.cpp** :

```
void Point::afficher()
{
printf("%d,%d\t",x,y);
}

void Point::positionner(int X,int Y)
{
x=X;
y=Y;
```

```
}
```

Dans cette écriture, il faut comprendre que l'identifiant de la fonction **`afficher(...)`** se rapporte à la classe Point (à son espace de noms, en fait, mais cela revient au même). C'est la même technique pour la méthode **`positionner()`**.

Nous avions déjà rencontré l'opérateur de résolution de portée **`::`** lorsque nous voulions atteindre une variable globale masquée par une variable locale à l'intérieur d'une fonction. En voilà une autre utilisation et ce n'est pas la dernière.

De nos jours, les compilateurs C++ sont très rapides et il n'y a plus de différence de performances à utiliser la définition complète de la classe lors de la déclaration ou bien la définition déportée dans un fichier **`.cpp`**. Les développeurs Java préféreront vraisemblablement la première version qui coïncide avec leur manière d'écrire une classe, mais l'approche déportée du C++, en plus d'être standard, offre comme avantage une lisibilité accrue si votre classe est assez longue. En effet, seule la connaissance des champs et de la signature des méthodes est importante pour utiliser une classe, alors que l'implémentation est à la charge de celui qui a créé la classe.

2. Instanciation

S'il est une règle impérative en science des algorithmes, c'est bien celle des valeurs consistantes pour les variables. Ainsi l'écriture suivante est une violation de cette règle :

```
int x,y;
    y = 2*x;
```

La variable x n'étant pas initialisée, il n'est pas possible de prédire la valeur de y. Bien que certains compilateurs initialisent les variables à 0, ce n'est ni une règle syntaxique, ni très rigoureux. Donc, chaque variable doit recevoir une valeur avant d'être utilisée, et si possible dès sa création.

D'ailleurs, certains langages comme Java ou C# interdisent l'emploi du fragment de code ci-dessus, soulevant une erreur bloquante et interrompant le processus de compilation.

Que se passe-t-il à présent lorsqu'une instance de classe (structure) est créée ? Un jeu de variables tout neuf est disponible, n'attendant plus qu'une méthode vienne les affecter, ou bien les lire. Et c'est cette dernière éventualité qui pose problème. Ainsi, considérons la classe Point et l'instanciation d'un nouveau point, puis son affichage. Nous employons l'instanciation avec l'opérateur **`new`** car elle est plus explicite que l'instanciation automatique sur la pile.

```
Point*p;         // un point qui n'existe pas encore
  p=new Point;   // instanciation
  p->afficher(); // affichage erroné, trop précoce
```

En principe, la méthode **`afficher()`** compte sur des valeurs consistantes pour les champs de coordonnées x et y. Mais ces champs n'ont jusqu'à présent pas été initialisés. La méthode **`afficher()`** affichera n'importe quelles valeurs, violant à nouveau la règle énoncée ci-dessus.

Comment alors initialiser au plus tôt les champs d'une nouvelle instance pour éviter au programmeur une utilisation inopportune de ses classes ? Tout simplement en faisant coïncider l'instanciation et l'initialisation. Pour arriver à ce résultat, on utilise un **constructeur**, méthode spéciale destinée à initialiser tous les champs de la nouvelle instance.

Nous compléterons l'étude des constructeurs au chapitre suivant et en terminons avec l'instanciation, qui se révèle plus évoluée que pour les structures.

Pour ce qui est de la syntaxe, la classe constituant un prolongement des structures, ces deux entités partagent les mêmes mécanismes :

- réservation par **`malloc()`** ;

- instanciation automatique, sur la pile ;
- instanciation à la demande avec l'opérateur **new**.

Pour les raisons qui ont été évoquées précédemment, la réservation de mémoire avec la fonction **malloc()** est à proscrire, car elle n'invoque pas le constructeur. Les deux autres modes, automatique et par l'opérateur **new** sont eux, parfaitement applicables aux classes.

```
Point p, m;  // deux objets instanciés sur la pile
Point* t = new Point; // un objet instancié à la demande
```

3. Constructeur et destructeur

Maintenant que nous connaissons mieux le mécanisme de l'instanciation des classes, nous devons définir un ou plusieurs constructeurs. Il est en effet assez fréquent de trouver plusieurs constructeurs, distingués par leurs paramètres (signature), entraînant l'initialisation des nouvelles instances en fonction de situations variées. Si l'on considère la classe **chaine**, il est possible de prévoir un constructeur dit par défaut, c'est-à-dire ne prenant aucun argument, et un autre constructeur recevant un entier spécifiant la taille du tampon à allouer pour cette chaîne.

a. Constructeur

Vous l'aurez compris, le constructeur est une méthode puisqu'il s'agit d'un groupe d'instructions, nommé, utilisant une liste de paramètres. Quelles sont les caractéristiques qui le distinguent d'une méthode ordinaire ? Tout d'abord le type de retour, car le constructeur ne retourne rien, pas même **void**. Ensuite, le constructeur porte le nom de la classe. Comme C++ est sensible à la casse - il différencie les majuscules et les minuscules - il faut faire attention à nommer le constructeur **Point()** pour la classe **Point** et non **point()** ou **Paint()**.

Comme toute méthode, un constructeur peut être déclaré dans le corps de la classe et défini de manière déportée, en utilisant l'opérateur de résolution de portée.

```
class Chaine
{
private:
  char*buffer;
  int t_buf;
  int longueur;

public:
  // un constructeur par défaut
  Chaine()
  {
    t_buf = 100;
    buffer = new char[t_buf];
    longueur = 0;
  }

  // un autre constructeur, défini hors de la classe
  Chaine(int taille);
} ;

Chaine::Chaine (int taille)
{
  t_buf = taille;
  buffer = new char[t_buf];
  longueur = 0;
}
```

Vous aurez bien noté le type de retour du constructeur : aucun type n'est spécifié. Le choix d'une implémentation au sein même de la classe ou alors hors d'elle est le vôtre, il suit la même logique que celle applicable aux méthodes ordinaires.

b. Le pointeur this

Il n'est pas rare qu'un constructeur reçoive un paramètre à répercuter directement sur un champ. Dans l'exemple précédent, l'argument taille a servi à initialiser le champ **t_buf**. Pourquoi alors s'embarrasser d'un nom différent si les deux variables désignent la même quantité ?

À cause du phénomène de masquage, répondrez-vous à juste titre :

```
Chaine::Chaine (int t_buf)
{
  t_buf = t_buf; // erreur
  buffer = new char[t_buf];
  longueur = 0;
}
```

La ligne normalement chargée d'initialiser le champ **t_buf** affecte l'argument **t_buf** en lui attribuant sa propre valeur. L'affaire se complique lorsqu'on apprend que l'opérateur de résolution de portée ne permet pas de résoudre l'ambiguïté d'accès à une variable de plus haut niveau comme nous l'avons fait pour le cas des variables globales. Il faut alors recourir au pointeur **this**, qui désigne toujours l'adresse de l'instance courante :

```
Chaine::Chaine (int t_buf)
{
  this->t_buf = t_buf;  // correct
  buffer=new char[t_buf];
  longueur=0;
}
```

Le pointeur **this** est toujours **Type***, où **Type** représente le nom de la classe. Dans notre constructeur de classe **Chaine**, **this** est un pointeur sur **Chaine**, soit un **Chaine***.

Signalons au passage que **this** peut atteindre n'importe quelle variable de la classe, quelle que soit sa visibilité, mais pas les champs hérités de classes supérieures lorsqu'ils sont privés. Par ailleurs, il est inutile d'utiliser **this** lorsqu'il n'y a pas d'équivoque entre un paramètre, une variable locale et un champ de la classe.

Enfin, **this** peut être utilisé pour transmettre l'adresse de l'objet courant à une fonction ou une méthode. Comme application très indirecte, on peut demander au constructeur d'afficher l'adresse du nouvel objet :

```
printf("Adresse %x\n",(int) this);
```

c. Destructeur

Puisque le constructeur a un rôle d'initialisation, il semblait logique de prévoir une méthode chargée de rendre des ressources consommées par un objet avant sa destruction.

Rappelons que la destruction de l'objet intervient soit à la demande du compilateur, lorsque le flot d'exécution quitte la portée d'une fonction ou d'un bloc ayant alloué automatiquement des objets, soit lorsque le programme reçoit l'ordre de destruction au moyen de l'opérateur **delete**.

```
void destruction()
{
Point* p;
Point m;  // instanciation automatique
  p = new Point; // instanciation manuelle
  printf("deux objets en cours");
  delete p; // destruction de p
  return;   // destruction de m implicite
}
```

Le destructeur ne reçoit jamais d'arguments, ce qui est tout à fait normal. Comme il peut être invoqué automatiquement, quelles valeurs le compilateur fournirait-il à cette méthode ?

Comme le constructeur, il ne renvoie rien, pas même **void** et porte le nom de la classe. En fait, il est reconnaissable par la présence du symbole ~ (tilde) placé avant son nom :

```
~Chaine()
{
  delete buffer;
}
```

Comme pour le constructeur, la syntaxe n'exige pas de doter chaque classe d'un destructeur. Sa présence est d'ailleurs liée à d'éventuelles réservations de mémoire ou de ressources système maintenues par l'objet.

La présence du constructeur n'est pas non plus imposée par la syntaxe, mais nous avons vu que son absence conduisait à un non-sens algorithmique.

Enfin, comme toute méthode, le destructeur peut être défini dans le corps de la classe ou bien de manière déportée, la syntaxe ressemblant alors à cela :

```
Chaine::~Chaine()
{
  delete buffer;
}
```

d. Destructeur virtuel

Bien que nous ne connaissions pas encore la dérivation (et l'héritage), ni les méthodes virtuelles, vous devez savoir que les destructeurs ont tout intérêt à être virtuels si la classe est héritée. Sans trop anticiper sur notre étude, prenez en compte le fait que les constructeurs sont toujours virtuels. Nous illustrerons le concept des méthodes virtuelles et des destructeurs virtuels un peu plus loin dans ce chapitre. Entre temps, voici la syntaxe de déclaration d'un tel destructeur :

```
virtual ~Chaine()
{
delete buffer;
}
```

Certains environnements de développement comme Visual C++ (Microsoft) déclarent systématiquement un constructeur et un destructeur virtuel pour une classe.

4. Allocation dynamique

Intéressons-nous maintenant à ce type de programmation appelé programmation dynamique. Comme les classes constituent un prolongement des structures, il est tout à fait possible de définir un pointeur de type **Classe*** pour construire des structures dynamiques, telles les listes, les arbres et les graphes. Nous vous proposons un exemple de classe **Liste** pour illustrer ce principe en même temps que les méthodes statiques.

D'autre part, vous devez faire attention lorsque vous allouez un tableau d'objets. Imaginons le scénario suivant :

```
Chaine*chaines;
  chaines=new Chaine[10];
```

Quel est le fonctionnement attendu par l'opérateur **new** ? Allouer 10 fois la taille de la classe **Chaine** ? Doit-il aussi appeler chaque constructeur pour chacune des 10 chaînes nouvellement créées ?

Pour déterminer son comportement, ajoutons un message dans le corps du constructeur :

```
class Chaine
{
private:
  char* buffer;
  int t_buf;
  int longueur;
public:
  // un constructeur par défaut
  Chaine()
  {
    printf("Constructeur par défaut, Chaine()\n");
    t_buf = 100;
    buffer = new char[t_buf];
    longueur=0;
  }

  Chaine (int t_buf)
  {
    printf("Constructeur Chaîne(%d)\n", t_buf);
    this->t_buf = t_buf;
    buffer = new char[t_buf];
    longueur = 0;
  }
} ;

int main(int argc, char* argv[])
{
  Chaine*chaines;
  chaines=new Chaine[10];
  return 0;
}
```

Testons le programme et analysons les résultats : le constructeur est bien appelé 10 fois. Comment dès lors appeler le deuxième constructeur, celui qui prend un entier comme paramètre ?

Vous ne pouvez pas combiner la sélection d'un constructeur avec paramètre et la spécification d'un nombre d'objets à allouer avec l'opérateur **new**. Il faut procéder en deux étapes, puisque l'écriture suivante n'est pas valide :

```
Chaines = new Chaine(75)[10];
```

Optez alors pour la tournure suivante, même si elle paraît plus complexe :

```
Chaine** chaines;
  chaines = new Chaine*[10];
  for(int i=0; i<10; i++)
    chaines[i] = new Chaine(75);
```

Attention, vous aurez remarqué le changement de type de la variable chaines, passée de **Chaine*** à **Chaine****. Ce changement diffère l'instanciation de l'allocation du tableau, devenu un simple tableau de pointeurs.

5. Constructeur de copie

Nous avons découvert lors de l'étude des structures que la copie d'une instance vers une autre, par affectation, avait pour conséquence la recopie de toutes les valeurs de la première instance vers la seconde. Cette règle reste vraie pour les classes.

```
Chaine s(20);
Chaine r;
Chaine x = s; // initialisation de x avec s
```

```
r = s;        // recopie s dans r
```

Nous devons également nous rappeler que cette copie pose des problèmes lorsque la structure (classe) possède des champs de type pointeur. En effet, l'instance initialisée en copie voit ses propres pointeurs désigner les mêmes zones mémoire que l'instance source. D'autre part, lorsque le destructeur est invoqué, les libérations à répétition des mêmes zones mémoire auront probablement un effet désastreux pour l'exécution du programme. Lorsqu'un bloc a été libéré, il est considéré comme perdu et libre pour une réallocation, donc il convient de ne pas insister.

Les classes de C++ autorisent une meilleure prise en charge de cette situation. Il faut tout d'abord écrire un constructeur dit de copie, pour régler le problème d'initialisation d'un objet à partir d'un autre. Ensuite, la surcharge de l'opérateur (cf. Autres aspects de la POO - Surcharge d'opérateurs) se charge d'éliminer le problème de la recopie par affectation.

```
class Chaine
{
// ... la déclaration ci-dessus
public:
  Chaine(const Chaine&);  // constructeur de copie
  Chaine& operator = (const Chaine&); // affectation copie
} ;
```

Commençons par le constructeur de copie :

```
Chaine::Chaine(const Chaine & ch)
{
  t_buf = ch.t_buf;
  longueur = ch.longueur;
  buffer = new char[ch.t_buf];   // ch référence
  for(int i=0; i<ch.longueur; i++)
    buffer[i]=ch.buffer[i];
}
```

Le constructeur de copie prend comme unique paramètre une référence vers un objet du type considéré.

Poursuivons avec la surcharge de l'opérateur d'affectation :

```
Chaine& Chaine::operator=(const Chaine& ch)
{
  if(this != &ch)
  {
    delete buffer;
    buffer = new char[ch.t_buf];   // ch référence
    for(int i=0; i<ch.longueur; i++)
      buffer[i] = ch.buffer[i];
    longueur = ch.longueur;

  }
  return *this;
}
```

Il faut faire attention à ne pas libérer par erreur la mémoire lorsque le programmeur procède à une affectation d'un objet vers lui-même :

```
s = s;
```

Pour une classe munie de ces deux opérations, les recopies par initialisation et par affectation ne devraient plus poser de problèmes. Naturellement, le programmeur a la charge de décrire l'algorithme approprié à la recopie.

Héritage

1. Dérivation de classe (héritage)

Maintenant que nous connaissons bien la structure et le fonctionnement d'une classe, nous allons rendre nos programmes plus génériques. Il est fréquent de décrire un problème général avec des algorithmes appropriés, puis de procéder à de petites modifications lorsqu'un cas similaire vient à être traité.

La philosophie orientée objet consiste à limiter au maximum les macros, les inclusions, les modules. Cette façon d'aborder les choses présente de nombreux risques lorsque la complexité des problèmes vient à croître. La programmation orientée objet s'exprime plutôt sur un axe générique/spécifique bien plus adapté aux petites variations des données d'un problème. Des méthodes de modélisation s'appuyant sur UML peuvent vous guider pour construire des réseaux de classes adaptés aux circonstances d'un projet. Mais il faut également s'appuyer sur des langages supportant cette approche, et C++ en fait partie.

a. Dérivation de classe (héritage)

Imaginons une classe **Compte**, composée des éléments suivants :

```
class Compte
{
protected:
  int numero;    // numéro du compte
  double solde;    // solde du compte

  static int num;  // variable utilisée pour calculer le prochain numéro
  static int prochain_numero();

public:
  char*titulaire;    // titulaire du compte

  Compte(char*titulaire);
  ~Compte(void);
  void crediter(double montant);
  bool debiter(double montant);
  void relever();
};
```

Nous pouvons maintenant imaginer une classe **CompteRemunere**, spécialisant le fonctionnement de la classe **Compte**. Il est aisé de concevoir qu'un compte rémunéré admet globalement les mêmes opérations qu'un compte classique, son comportement étant légèrement modifié pour ce qui est de l'opération de crédit, puisqu'un intérêt est versé par l'organisme bancaire. En conséquence, il est fastidieux de vouloir réécrire totalement le programme qui fonctionne pour la classe **Compte**. Nous allons plutôt dériver cette dernière classe pour obtenir la classe **CompteRemunere**.

La classe **CompteRemunere**, quant à elle, reprend l'ensemble des caractéristiques de la classe **Compte** : elle hérite de ses champs et de ses méthodes. On dit pour simplifier que **CompteRemunere** hérite de **Compte**.

```
class CompteRemunere : public Compte
{
protected:
  double taux;

public:
  CompteRemunere(char*titulaire,double taux);
  ~CompteRemunere(void);

  void crediter(double montant);
};
```

Vous aurez remarqué que seules les différences (modifications et ajouts) sont inscrites dans la déclaration de la classe qui hérite. Tous les membres sont transmis par l'héritage **`: public Compte`**.

Voici maintenant l'implémentation de la classe **`Compte`** :

```
#include "compte.h"
#include <string.h>
#include <stdio.h>

Compte::Compte(char*titulaire)
{
  this->titulaire=new char[strlen(titulaire)+1];
  strcpy(this->titulaire,titulaire);
  solde=0;
  numero=prochain_numero();
}

Compte::~Compte(void)
{

  delete titulaire;
}

void Compte::crediter(double montant)
{
  solde+=montant;
}

bool Compte::debiter(double montant)
{
  if(montant<=solde)
  {
    solde-=montant;
    return true;
  }
  return false;
}

void Compte::relever()
{
  printf("%s (%d) : %.2g euros\n",titulaire,numero,solde);
}

int Compte::num=1000;

int Compte::prochain_numero()
{
  return num++;
}
```

Mis à part l'utilisation d'un champ et d'une méthode statiques, il s'agit d'un exercice connu. Passons à la classe **`CompteRemunere`** :

```
#include ".\compteremunere.h"

CompteRemunere::CompteRemunere(char*titulaire,
                double taux) : Compte(titulaire)

{
  this->taux=taux;
}

CompteRemunere::~CompteRemunere(void)
{
}

void CompteRemunere::crediter(double montant)
```

```
{
  Compte::crediter(montant*(1+taux/100));
}
```

L'appel du constructeur de la classe **Compte** depuis le constructeur de la classe **CompteRemunere** sera traité dans un prochain paragraphe. Concentrons-nous plutôt sur l'écriture de la méthode **crediter**.

Cette méthode existe déjà dans la **classe de base**, **Compte**. Mais il faut remarquer que le crédit sur un compte rémunéré ressemble beaucoup au crédit sur un compte classique, sauf que l'organisme bancaire verse un intérêt. Nous devrions donc appeler la méthode de base, **crediter**, située dans la classe **Compte**. Hélas, les règles de visibilité et de portée font qu'elle n'est pas accessible autrement qu'en spécifiant le nom de la classe à laquelle on fait référence :

```
void CompteRemunere::crediter(double montant)
{
  // portée la plus proche => récurrence infinie
  crediter(montant*(1+taux/100));

  // écriture correcte
  Compte::crediter(montant*(1+taux/100));
}
```

Les programmeurs Java et C# ne connaissant pas l'héritage multiple, on a substitué le nom de la classe de base par un mot clé unique, respectivement **super** et **base**.

Pour étudier le fonctionnement du programme complet, nous fournissons le module principal, **main.cpp** :

```
//#include "compte.h" // inutile car déjà présent dans compteremunere.h
#include "compteremunere.h"

int main(int argc, char* argv[])
{
  Compte c1("Victor Hugo");
  c1.crediter(100);
  c1.relever();
  c1.debiter(80);
  c1.relever();

  CompteRemunere r1("Jean Valjean",5);
  r1.crediter(100);
  r1.relever();
  return 0;
}
```

L'exécution donne les résultats suivants :

Le crédit de 100 euros sur le compte de Jean Valjean a bien été rémunéré à hauteur de 5 %.

b. Héritage public, protégé et privé

Il existe une spécificité du langage C++ : il est possible de restreindre l'accès aux membres de la classe dérivée en contrôlant la publicité de l'héritage.

```
class Derive : public Base
{
...
};
```

Le mot clé **`public`** peut être remplacé par **`private`** ou **`protected`**. Le tableau suivant énumère les cas d'accessibilité des champs d'une classe de base depuis une classe dérivée, en fonction de leur niveau de protection initial et du type de dérivation.

Visibilité dans la classe de base	Type de dérivation	Visibilité dans la classe dérivée
public	public	public
protected	public	protected
private	public	inaccessible
public	private	private
protected	private	private
private	private	private
public	protected	private
protected	protected	protected
private	protected	protected

Nous vous recommandons toutefois de ne pas abuser de ce type de construction, car les langages Java et C# ne disposent que de l'héritage public.

c. Appel des constructeurs

Nous savons qu'une classe dérivant d'une autre classe hérite de l'ensemble de ses champs, soient-ils privés. C'est tout à fait normal, car une classe est toujours décrite comme un tout. Elle a besoin de l'ensemble de ses champs pour adosser son algorithmie, aussi une classe n'est jamais décrite avec l'arrière-pensée de la dériver par la suite. Ce motif de conception est réservé aux classes abstraites, c'est-à-dire contenant au moins une méthode virtuelle pure et peu d'algorithmie.

De ce fait, il faut trouver un moyen d'initialiser l'ensemble des champs, y compris ceux provenant de la classe de base, privés ou non. Ce travail d'initialisation a déjà été réalisé à l'aide d'un ou de plusieurs constructeurs.

D'autre part, c'est bien la classe héritant qui est instanciée, et son constructeur appelé pour initialiser ses propres champs. Il paraît dès lors légitime de chercher à appeler au plus tôt un constructeur dans la classe de base, ou les constructeurs des classes de base en cas d'héritage multiple.

Le concepteur de C++ a voulu souligner que cet appel devait figurer avant l'exécution de la première instruction du constructeur de la classe héritant, aussi a-t-il prévu une syntaxe particulière :

```
CompteRemunere::CompteRemunere(
char*titulaire, double taux) : Compte(titulaire)
{
```

```
  this->taux=taux;
}
```

Le constructeur de la classe **Compte** est appelé avant l'affectation du champ taux. En cas d'héritage multiple, on sépare les constructeurs par des virgules :

```
ConstructeurHerite(params) : Constructeur1(params),  Constructeur2(params)
{
}
```

Vous aurez noté la possibilité de transmettre au constructeur de la classe supérieure des paramètres reçus dans la classe héritant, ce qui est tout à fait légitime.

Si votre classe de base n'a pas de constructeur, ou si elle en possède un par défaut (sans argument), le compilateur peut réaliser cet appel automatiquement. Toutefois, l'absence de constructeur ou l'absence de choix explicite d'un constructeur constituent une faiblesse dans la programmation, et probablement dans la conception.

2. Polymorphisme

Le polymorphisme caractérise les éléments informatiques existant sous plusieurs formes. Bien qu'il ne s'agisse pas à proprement parler d'une caractéristique nécessaire aux langages orientés objet, nombre d'entre eux la partagent.

a. Méthodes polymorphes

Pour définir une méthode polymorphe, il suffit de proposer différents prototypes avec autant de signatures (listes d'arguments). La fonction doit toujours porter le même nom, autrement elle ne serait pas polymorphe. Le compilateur se base sur les arguments transmis lors de l'invocation d'une méthode, en nombre et en types, pour déterminer la version qu'il doit appeler.

```
class Chaine
{
...
  void concat(char c);
  void concat(Chaine s);
  void concat(int nombre);
} :
```

Faites attention lorsqu'une méthode polymorphe est invoquée à l'aide d'une valeur littérale comme paramètre, il y a parfois des ambiguïtés délicates à résoudre. Le transtypage (changement de type par coercition) apparaît alors comme l'une des solutions les plus claires qui soit.

Dans notre exemple de classe Chaine, nous allons invoquer la méthode **concat()** avec la valeur littérale 65. Si 65 est le code ASCII de la lettre B, c'est aussi un entier. Quelle version choisit le compilateur ?

```
Chaine s="abc";
s.concat(65);
```

Suivant la version appelée, nous devrions obtenir soit la chaîne "abcA" soit la chaîne "abc65".

Nous pouvons résoudre l'équivoque à l'aide d'un transtypage adapté aux circonstances :

```
s.concat( (char) 65 );
```

Nous sommes certains cette fois que c'est la version recevant un char qui est appelée. Il vaut mieux être le plus explicite possible car le compilateur C++ est généralement très permissif et très implicite dans ses décisions !

b. Conversions d'objets

S'il est une méthode qui est fréquemment polymorphe, c'est certainement le constructeur. Celui-ci permet d'initialiser un objet à partir d'autres objets :

```
Chaine x('A');
Chaine t(x);
```

Lorsque nous aurons étudié la surcharge des opérateurs, nous verrons comment tirer parti de cette multitude de constructeurs offerts à certaines classes.

3. Méthodes virtuelles et méthodes virtuelles pures

Le langage C++ est un langage compilé, et de ce fait, fortement typé. Cela signifie que le compilateur suit au plus près les types prévus pour chaque variable.

Livrons-nous à l'expérience suivante : considérons deux classes, **Base** et **Derive**, la seconde dérivant naturellement de la première. La classe **Base** ne contient pour l'instant qu'une méthode, **methode1()**, surchargée (réécrite) dans la seconde classe :

```
class Base
{
public:
  void methode1()
  {
    printf("Base::methode1\n");
  }
}  ;

class Derive : public Base
{
public:
  void methode1()
  {
    printf("Derive::methode1\n");
  }
} ;
```

Nous proposons maintenant une fonction **main()** instanciant chacune de ces classes dans le but d'invoquer la méthode **methode1()**.

```
int main(int argc, char* argv[])
{
  Base*b;
  Derive*d;
  b = new Base();
  b->methode1();

  d = new Derive();
  d->methode1();
  return 0;
}
```

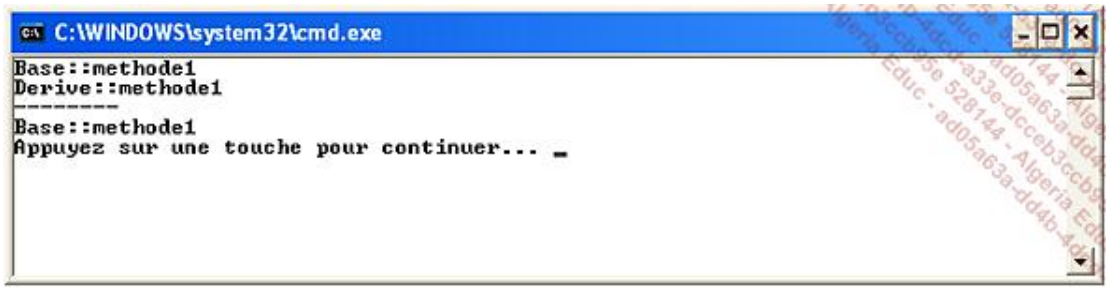
L'exécution du programme fournit des résultats en concordance avec les instructions contenues dans la fonction **main()** :

Le compilateur a suivi le typage des pointeurs **b** et **d** pour appliquer la bonne méthode.

Maintenant considérons la classe **Base** comme un sous-ensemble de la classe **Derive**. Il est légal d'écrire **b=d** puisque toutes les opérations applicables à **b** sont disponibles dans **d**, l'opération est sans risque pour l'exécution. Quelle va être alors le choix du compilateur lorsque nous écrirons **b->methode1()** ?

```
printf("--------\n");
b = d;
b->methode1();
```

L'exécution du programme nous fournit le résultat suivant, le compilateur ayant finalement opté pour la version proposée par **Base**.



Ce résultat est tout à fait logique, en tenant compte du fait que l'édition des liens a lieu avant l'exécution. Aucun mécanisme n'est appliqué pour retarder l'édition des liens et attendre le dernier moment pour déterminer quel est le type désigné par le pointeur **b**.

Ce mécanisme existe dans C++, il s'agit des méthodes virtuelles.

Complétons à présent nos classes par **methode2()**, dont le prototype est marqué par le mot clé **virtual** :

```
class Base
{
public:
  void methode1()
  {
    printf("Base::methode1\n");
  }

  virtual void methode2()
  {
    printf("Base::methode2\n");
  }
} ;

class Derive : public Base
{
public:
  void methode1()
  {
    printf("Derive::methode1\n");
  }
```

```
  void methode2()
  {
    printf("Derive::methode2\n");
  }
} ;
```

Vous aurez noté qu'il n'est pas nécessaire de marquer la version héritière, l'attribut **virtual** est automatiquement transmis lors de la dérivation.

Un nouveau **main()** fournit des résultats différents :

```
printf("--------\n");
b = d;
b->methode2();
```

La seconde méthode étant virtuelle, le compilateur retarde l'éditeur de liens. C'est à l'exécution que l'on détermine quel est le type désigné par le pointeur **b**. L'exécution est plus lente mais le compilateur appelle "la bonne version".

En fait, les méthodes virtuelles représentent la grande majorité des besoins des programmeurs, aussi le langage Java a-t-il pris comme parti de supprimer les méthodes non virtuelles. De ce fait, le mot clé **virtual** a disparu de sa syntaxe. Le langage C# est resté plus fidèle au C++ en prévoyant les deux systèmes mais en renforçant la syntaxe de surcharge, afin que le programmeur exprime son intention lorsqu'il dérive une classe et surcharge une méthode.

Les méthodes virtuelles sont particulièrement utiles pour programmer les interfaces graphiques et les collections. Signalons qu'elles fonctionnent également avec des références :

```
Base& br = *d;
br.methode1();
br.methode2();
```

Il arrive fréquemment qu'une méthode ne puisse être spécifiée complètement dans son algorithmie, soit parce que l'algorithmie est laissée aux soins du programmeur, soit parce que la classe exprime un concept ouvert, à compléter à la demande.

Les méthodes virtuelles pures sont des méthodes qui n'ont pas de code. Il faudra dériver la classe à laquelle elles appartiennent pour implémenter ces méthodes :

```
class Vehicule
{
public:
  virtual void deplacer() =0 ;
} ;

class Voiture : public Vehicule
{
public:
  void deplacer()
  {
    printf("vroum");
  }
} ;
```

La méthode virtuelle pure est signalée par la conjonction de la déclaration **virtual** et par l'affectation d'un pointeur nul (**=0**). De ce fait, la classe **Vehicule**, abstraite, n'est pas directement instanciable. C'est **Voiture** qui est chargée d'implémenter tout ou partie des méthodes virtuelles pures reçues en héritage. S'il reste au moins une méthode virtuelle pure n'ayant pas reçu d'implémentation, la classe **Voiture** reste abstraite. Ce n'est certes pas le cas de notre exemple.

```
Vehicule*veh = new Vehicule; // interdit car classe abstraite
```

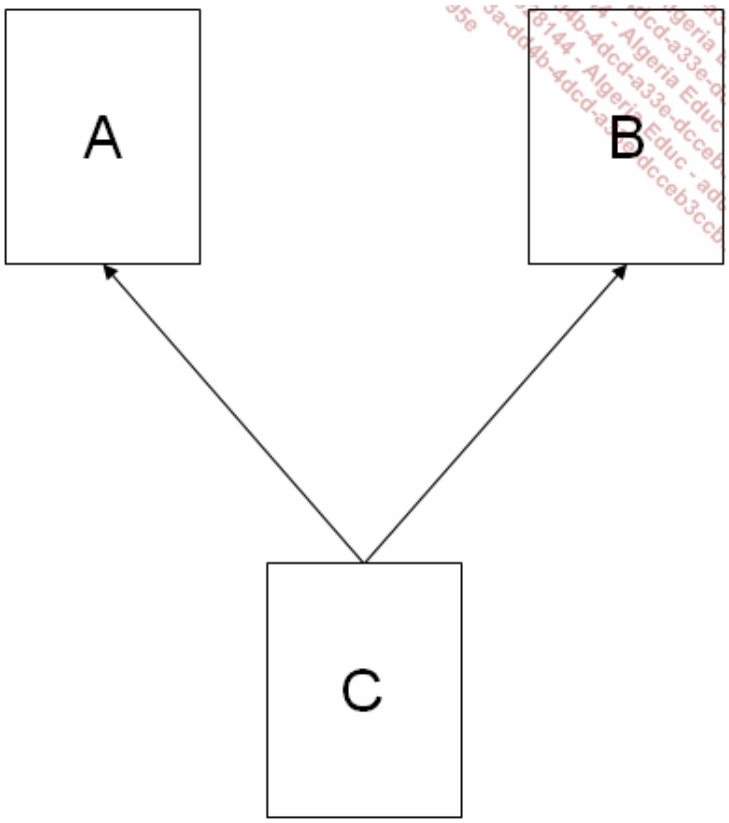
```
Voiture *voit1 = new Voiture; // ok
voit1->deplacer;
veh = new Voiture; // ok
```

Le langage Java ne connaissant pas les pointeurs, il a remplacé l'écriture **=0** ou **=NULL** par le mot clé **abstract**. Cela signifie que les langages C++ et Java (en C#) peuvent exprimer les mêmes notions objet, ce qui est déterminant lorsque l'implémentation suit une modélisation comme UML.

Les langages successeurs de C++ proposent également la notion d'interface : il s'agit d'une classe abstraite dont toutes les méthodes sont abstraites, autrement dit virtuelles pures dans le vocabulaire C++. L'interface a sans aucun doute été proposée pour simplifier l'écriture des programmes, mais elles ne constituent pas une nouveauté en tant que telle. C++ est donc capable d'exprimer des modélisations faisant intervenir des interfaces.

4. Héritage multiple

Une situation inconfortable s'est présentée aux concepteurs de programmes orientés objet : une classe devait spécialiser plusieurs concepts simultanément. De ce fait, elle devait dériver d'au moins deux classes, et hériter de chacun de ses membres. L'auteur de C++ a pourvu son langage d'une syntaxe respectant cette modélisation que l'on désigne sous l'appellation héritage multiple.



Dans notre exemple, classe C hérite à la fois de la classe A et de la classe B.

En termes C++, nous aurions traduit cette modélisation de la façon suivante :

```
class A { ... } ;
class B { ... } ;
class C : public A, public B
{
...
} ;
```

a. Notations particulières

La première notation concerne naturellement l'appel des constructeurs. La classe C héritant des classes A et B, elle doit appeler les deux constructeurs, dans l'ordre de son choix. En fait, l'ordre d'initialisation n'a normalement pas d'importance, une classe étant un tout, elle n'est pas conçue en fonction d'autres classes. Si l'ordre revêt une importance particulière pour que le programme fonctionne, la modélisation est probablement à revoir.

```
C(param) : A(params), B(params)
{
...
}
```

Évidemment, les paramètres applicables aux trois constructeurs peuvent être différents en valeur, en type et en nombre.

Par la suite, nous devons distinguer les méthodes qui portent le même nom dans les classes A et B, et qui seraient invoquées par une méthode de C.

Imaginons la situation suivante :

```
class A
{
  void methode1() { ... }
} ;
```

et

```
class B
{
  void methode1() { ... }
} ;
```

Que se passe-t-il si une méthode de C appelle **`methode1()`**, dont elle a en fait hérité deux fois ? En l'absence d'un marquage particulier, le compilateur soulève une erreur car l'appel est ambigu :

```
class C : public A, public B
{
  void methode2()
  {
    methode1() ; // appel ambigu
  }
} ;
```

Lorsque de tels cas se présentent, le programmeur doit faire précéder le nom de la méthode du nom de la classe à laquelle il fait référence. C'est bien entendu l'opérateur de résolution de portée qui intervient à ce moment- là :

```
class C : public A, public B
{
  void methode2()
  {
    A ::methode1() ; // ok
  }
} ;
```

À l'extérieur de la classe, le problème demeure, car le compilateur ne sait pas quelle version appeler :

```
class A
```

```
{
public:
  int f(int i)
  {
    printf("A::f(int=%d)\n",i);
    return i;
  }
  char f(char i)
  {
    printf("A::f(char=%c)\n",i);
    return i;
  }
} ;

class B
{
public:
  double f(double d)
  {
    printf("B::f(double=%g)\n",d);
    return d;
  }
} ;
class AetB : public A, public B
{
public:
  char f(char a)
  {
    printf("AetB::f(char=%c)\n",a);
    return a;
  }
  bool f(bool b)
  {
    printf("AetB::f(boolean=%d)\n",b);
    return b;
  }
} ;
int main(int argc, char* argv[])
{
  AetB x;
  x.f(1);    // appel ambigu
  x.f(2.0);    // appel ambigu
  x.f('A');
  x.f(true);
  return 0;
}
```

Il existe deux moyens pour lever l'ambiguïté des appels signalés dans la fonction **`main()`**. Le premier consiste à faire précéder l'invocation de f par **`A::`** ou **`B::`**, comme indiqué :

```
x.A::f(1000);
x.B::f(2.0);
```

Le second moyen consiste à utiliser dans le corps de la déclaration de la classe AetB des déclarations d'utilisation : **`using`**

```
using A::f;
using B::f;
```

Le second moyen est à privilégier car il augmente la qualité de la modélisation.

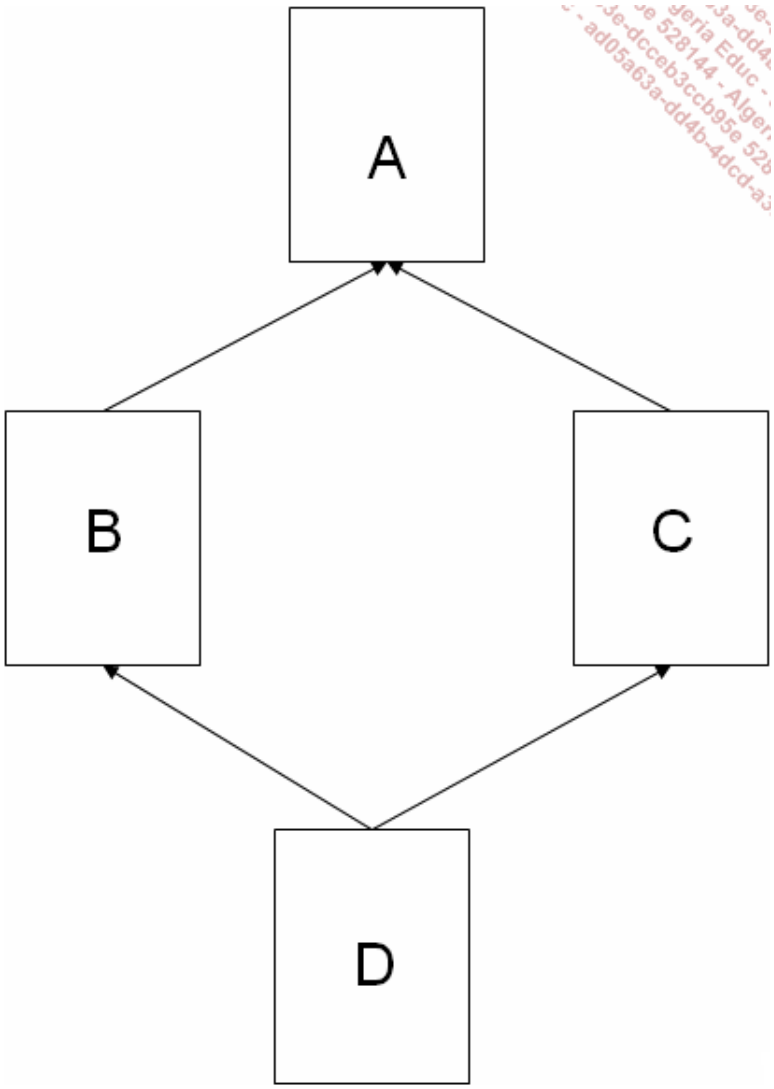
b. Conséquences sur la programmation

Cette caractéristique a fortement augmenté la complexité des compilateurs, en même temps que la vie des

programmeurs. Imaginons une classe A dérivant en deux classes B et C. Ajoutons à notre diagramme une classe D qui hérite simultanément de B et de C. Elle devrait hériter deux fois des membres de A.

Les langages Java et C# ont supprimé l'héritage multiple, en autorisant seulement l'héritage d'une classe et l'implémentation d'une interface. Une interface étant abstraite, elle ne contient ni champ, ni code. Les doutes sont par conséquent levés.

L'auteur de C++ a lui proposé le mécanisme des classes virtuelles.



En principe, tous les membres de la classe A sont hérités deux fois par la classe D. L'utilisation de l'opérateur de résolution de portée **::** contrôle l'accès à ces membres. Lorsque le concepteur (ou le programmeur) ne veut qu'un seul héritage, il peut utiliser le mécanisme de la dérivation virtuelle :

```
class B : public virtual A
{ ... } ;
class C : public virtual A
{ ... } ;

class D : public virtual B, public virtual C
{ ... } ;
```

Le compilateur s'assurera alors que chaque membre n'est hérité qu'une fois. Toutefois, le programmeur ne pourra pas différencier l'origine des membres hérités. Ce mécanisme est donc à manier avec prudence.

Autres aspects de la POO

1. Conversion dynamique

a. Conversions depuis un autre type

Les constructeurs permettent de convertir des objets à partir d'instances (de valeurs) exprimées dans un autre type.

Prenons le cas de notre classe **`Chaine`**. Il serait intéressant de "convertir" un **`char*`** ou un **`char`** en chaîne :

```
#include <string.h>

class Chaine
{
private:
  char*buffer;
  int t_buf;
  int longueur;
public:
  // un constructeur par défaut
  Chaine()
  {
    t_buf=100;
    buffer=new char[t_buf];
    longueur=0;
  }
  Chaine (int t_buf)
  {
    this->t_buf=t_buf;
    buffer=new char[t_buf];
    longueur=0;
  }

Chaine(char c)
  {
    t_buf=1;
    longueur=1;
    buffer=new char[t_buf];
    buffer[0]=c;
  }

  Chaine(char*s)
  {
    t_buf=strlen(s)+1;
    buffer=new char[t_buf];
    longueur=strlen(s);
    strcpy(buffer,s);
  }

  void afficher()
  {
    for(int i=0; i<longueur; i++)
      printf("%c",buffer[i]);
    printf("\n");
  }
} ;

int main(int argc, char* argv[])
{
  // Ecriture 1
  // Conversion par emploi explicite de constructeur
  Chaine x;
```

```
  x=Chaine("bonjour");  // conversion
  x.afficher();

  // Ecriture 2
  // Transtypage (cast) par coercition
  Chaine y=(char*) "bonjour"; // conversion (cast)
  y.afficher();

  // Ecriture 3
  // Transtypage (cast), approche C++
  Chaine z=char('A');    // conversion (cast)
  z.afficher();
  return 0;
}
```

La fonction **`main()`** illustre les trois façons de convertir vers un type objet :

- par l'emploi d'un constructeur explicite (1) ;
- par l'emploi d'un transtypage, comme en C (2) ;
- en utilisant la syntaxe propre à C++ (3).

Nous verrons que la surcharge de l'opérateur de conversion permet d'aller plus loin, notamment pour convertir un objet vers un type primitif. En conjonction avec la surcharge de l'opérateur d'affectation, il devient possible de réaliser pratiquement n'importe quel type de conversion, sans avoir à recourir à des méthodes spécialisées (**`ToInt`**, **`ToString`**, **`ToChar`**, **`FromInt`**, **`FromString`**...).

b. Opérateurs de conversion

Le langage C++ dispose de deux opérateurs de conversion spécifiques :

```
const_cast<type> expressionconst_cast
```

Convertit une expression comme le ferait (type) expression si le type de expression diffère uniquement par les modificateurs **`const`** ou **`volatile`**.

```
reinterpret_cast<type> expressionreinterpret_cast
```

Convertit un pointeur en un autre pointeur sans trop se soucier de la cohérence de l'opération.

Le premier opérateur, **`const_cast`**, sert à oublier provisoirement l'utilisation du modificateur **`const`**. Les fonctions constantes, marquées **`const`** après leur prototype, engagent à ne pas modifier les champs de la classe. Toutefois, cette situation peut être parfois gênante. **`const_cast`** autorise la conversion d'un pointeur de classe vers une classe identique, mais débarrassée des marqueurs **`const`**.

```
class CCTest
{
public:
   void setNombre( int );
   void afficheNombre() const;
private:
   int nombre;
};

void CCTest::setNombre(int num)
{
   nombre = num;
}

void CCTest::afficheNombre() const
{
```

```
   printf("Avant: %d",nombre);
   const_cast< CCTest * >( this )->nombre--;
   printf("Après %d",nombre);
}

int main()
{
   CCTest X;
   X.setNombre( 8 );
   X.afficheNombre();
}
```

C'est dans la méthode **afficheNombre** que l'interdiction a été provisoirement levée. Signalons également que cet opérateur ne permet pas directement de lever l'interdiction de modification sur des champs **const**.

Quant à l'opérateur **reinterpret_cast**, il se révèle potentiellement assez dangereux et ne devrait être utilisé que pour des fonctions de bas niveau, comme dans l'exemple du calcul d'une valeur de hachage basée sur la valeur entière d'un pointeur :

```
unsigned short Hash( void *p )
{
   unsigned int val = reinterpret_cast<unsigned int>( p );
   return ( unsigned short )( val ^ (val >> 16));
}
```

c. Conversions entre classes dérivées

Deux opérateurs supplémentaires contrôlent les conversions de type, tout particulièrement lorsqu'il s'agit de classes dérivées. Lors de l'étude des méthodes virtuelles, nous avons envisagé deux classes, **Base** et **Derive** la seconde dérivant de la première.

L'opérateur **static_cast** s'assure qu'il existe une conversion implicite entre les deux types considérés. De ce fait, il est dangereux de l'utiliser pour remonter un pointeur dans la hiérarchie des classes, c'est-à-dire d'utiliser un pointeur **Derive*** pour désigner un **Base***. Finalement, l'opérateur **static_cast**, qui donne la pleine mesure de son action lors de la compilation, est beaucoup plus utile pour réaliser des conversions propres entre des énumérations et des entiers.

```
enum Couleurs { rouge, vert, bleu };

int conv_couleur(Couleurs coul)
{
  return static_cast<int> ( coul );
}
```

Pour utiliser l'opérateur **dynamic_cast**, il faut parfois activer un commutateur du compilateur, comme **ENABLE_RTTI** (*Run Time Type Information*) pour le compilateur Microsoft. Il devient alors trivial d'envisager des conversions pour lesquelles un doute subsiste quant au type réellement manipulé : **dynamic_cast**.

```
void conie_sans_risque(Base*b)
{
  Derive*d1 = dynamic_cast<Derive*>(b);
}
```

Si l'argument désigne effectivement un objet de type **Base**, sa promotion directe en **Derive** (par coercition) est assez risquée. L'opérateur **dynamic_cast** renverra **NULL**, voire lèvera une exception. Toutefois, la détection devra attendre l'exécution pour déterminer le type exact représenté par **b**.

2. Champs et méthodes statiques

a. Champs statiques

Un champ statique est une variable placée dans une classe mais dont la valeur est indépendante des instances de cette classe : elle n'existe qu'en un seul et unique exemplaire. Quelle est l'opportunité d'une telle variable ? Tout d'abord, il existe des situations pour lesquelles des variables sont placées dans une classe par souci de cohérence.

Envisageons la classe **Mathematiques** et la variable **PI**. Il est logique de rattacher **PI** dans le vocabulaire mathématique, autrement dit de classer **PI** dans **Mathematiques**. Mais la valeur de **PI** est constante, et elle ne saurait de toute façon varier pour chaque instance de cette classe. Pour évoquer ce comportement, il a été choisi de marquer le champ **PI** comme étant **statique**.

Deuxièmement, les méthodes statiques (cf. partie suivante) n'ont pas accès aux variables d'instance, c'est-à-dire aux champs. Si une méthode statique a besoin de partager une valeur avec une autre méthode statique, il ne lui reste plus qu'à adresser une variable statique. Attention, il s'agit bien d'un champ de la classe et non d'une variable locale statique, disposition qui a heureusement disparu dans les langages de programmation actuels.

```
class Mathematiques
{
public :
  static double PI ;
} ;
double Mathematiques ::PI=3.14159265358 ;
```

Vous aurez noté que la déclaration d'un champ statique n'équivaut pas à son allocation. Il est nécessaire de définir la variable à l'extérieur de la classe, en utilisant l'opérateur de résolution de portée **::** pour réellement allouer la variable.

Comme autre exemple de champ statique, reportez-vous à l'exemple précédent de la classe **Compte**. Le prochain numéro de compte, un entier symbolisant un compteur, est incrémenté par une méthode elle-même statique.

Soulignons qu'il y a une forte similitude entre un champ statique, appartenant à une classe, et une variable déclarée dans un espace de noms, que d'aucuns appellent module.

b. Méthodes statiques

Les méthodes statiques reprennent le même principe que les champs statiques. Elles sont déclarées dans une classe, car elles se rapportent à la sémantique de cette classe, mais sont totalement autonomes vis-à-vis des champs de cette classe. Autrement dit, il n'est pas nécessaire d'instancier la classe à laquelle elles appartiennent pour les appliquer, ce qui signifie qu'on ne peut les appliquer à des objets.

Restons pour l'instant dans le domaine mathématique. La fonction **cosinus(angle)** peut être approchée à partir d'un développement limité, fonction mathématique très simple :

$$cosinus(angle) = \Sigma\ (angle_i\ /\ !i)$$

pour **i=2*k**, k variant de 0 à n.

Nous en déduisons pour C++ la fonction cosinus :

```
double cosinus(double a)
{
  return 1-a*a/2+a*a*a*a/24+a*a*a*a*a*a/720 ;
}
```

Nous remarquons que cette fonction est à la fois totalement autonome, puisqu'elle ne consomme qu'un argument **a**, et qu'elle se rapporte au domaine mathématique. Nous décidons d'en faire une méthode statique :

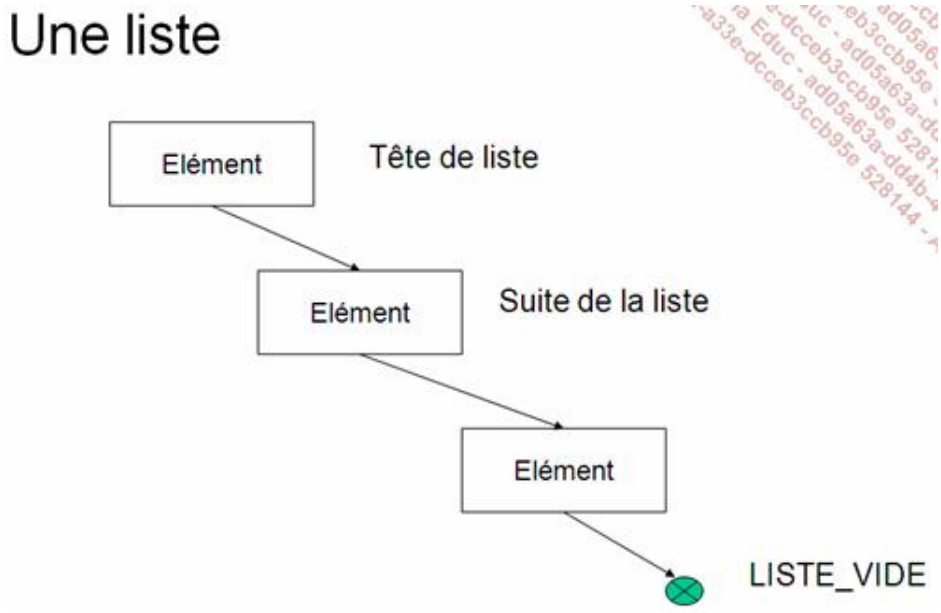
```
class Mathematiques
{
public :
double cosinus(double a)
{
  return 1-a*a/2+a*a*a*a/24+a*a*a*a*a*a/720 ;
}

} ;
```

Maintenant, pour appeler cette méthode, nous n'avons qu'à l'appliquer directement à la classe **Mathematiques** sans avoir besoin de créer une instance :

```
double angle=Mathematique ::PI/4 ;
  printf("cos(pi/4)=%g",Mathematiques::cosinus(a));
```

Le résultat approche $\sqrt{}$ 2/2, soit 0,707. Le microprocesseur ne s'y prend pas autrement pour fournir la valeur d'un cosinus à la fonction **cos()** disponible dans **<math.h>**. Certains processeurs numériques factorisent la variable **a** et utilisent une table de poids pour augmenter la rapidité du calcul.

Nous donnons maintenant un exemple de structure dynamique, la liste, qui a la faveur des enseignements en informatique. Cette structure stocke des éléments de nature diverse - entiers, chaînes... - et sa taille augmente au fur et à mesure des besoins.



On utilise en général les fonctions suivantes pour manipuler les listes : **cons()** qui sera notre constructeur, **suite()** et **est_vide()** qui teste la nullité d'une liste. Les applications des listes sont innombrables, et leur apprentissage est un plus pour tout programmeur.

Traditionnellement traitée en langage C, la liste gagne à être implémentée en C++, notamment par l'emploi de méthodes statiques. Voici donc l'implémentation proposée. Nous commençons par le fichier **Liste.h**, qui possède plusieurs membres statiques.

```
#pragma once

#include <stdio.h>

class Liste
{
```

```
public:
  char*element;
  Liste* _suite;
  static const Liste* LISTE_VIDE;

  Liste()
  {
    _suite=const_cast<Liste*>(LISTE_VIDE);
  }

  Liste(char*e,Liste*suite)
  {
    _suite=suite;
    element=e;
  }

  Liste* suite()
  {
    return _suite;
  }

  static bool est_vide(Liste* l)
  {
    return l==LISTE_VIDE;
  }

  void afficher()
  {
    Liste::afficher(this);
  }

  static void afficher(Liste*l)
  {
    if(Liste::est_vide(l))
      return;
    printf("%s,",l->element);
    Liste::afficher(l->suite());
  }
} ;
```

La fonction **est_vide** étant totalement autonome, il est logique de la définir statique. L'affichage d'une liste est grandement facilité par l'écriture d'une fonction récursive prenant une liste comme paramètre. De ce fait, la fonction ne s'applique à aucune instance en particulier et la méthode correspondante devient elle-même statique. L'exposition de cette méthode récursive avec le niveau public ou privé est un choix qui appartient au programmeur.

Nous trouvons ensuite dans le fichier **Liste.cpp** la définition du champ statique **LISTE_VIDE**. Notez au passage la combinaison des modificateurs **static** et **const**, combinaison fréquente s'il en est.

```
#include ".\liste.h"

const Liste* Liste::LISTE_VIDE=NULL;
```

La définition déportée de cette variable ne pouvait en aucune manière figurer dans le fichier **Liste.h**, car son inclusion par plusieurs modules **cpp** aurait entraîné sa duplication entre chaque module, rendant ainsi impossible l'édition des liens.

La fonction **main()** crée un exemple de liste puis appelle la méthode **afficher()** :

```
#include "Liste.h"


int main(int argc, char* argv[])
{
  Liste* liste=new Liste("bonjour",
```

```
        new Liste("les",
         new Liste("amis",
const_cast <Liste*> (Liste::LISTE_VIDE) )));
  liste->afficher();
  return 0;
}
```

Au passage, vous aurez noté l'emploi de l'opérateur **const_cast** pour accorder précisément les types manipulés.

3. Surcharges d'opérateurs

Le concepteur de C++ a voulu que son langage soit le plus expressif possible ; définir des classes et créer de nouveaux types est une chose, conserver une programmation simple et claire en est une autre. La surcharge des opérateurs, c'est-à-dire leur adaptation à des classes définies par le programmeur va justement dans cette direction.

Un opérateur est en fait assimilable à une fonction. Cette fonction réunit un certain nombre d'opérandes et évalue un résultat d'un type particulier. Pour qu'un opérateur soit surchargé, il doit figurer dans une classe comme n'importe quelle méthode.

a. Syntaxe

En fait, il est laissé une très grande liberté au programmeur dans le type des arguments applicables aux opérateurs surchargés. Le respect de la commutativité, des priorités, de la sémantique même d'un opérateur peuvent être remis en cause sans que le compilateur ne trouve à y redire.

La syntaxe est la suivante :

```
class Classe
{
      type_retour operator op (type_op operande)
      {
      ...
      }
} ;
```

Cet opérateur, **op**, est destiné à figurer entre une instance de la classe **Classe** et un opérande de type **type_op**. La fonction est libre de modifier l'instance à laquelle elle est appliquée et peut renvoyer un résultat de n'importe quel type, même **void**.

Pour illustrer cette notation, reprenons l'exemple de la classe **Chaine** vu précédemment. Ajoutons la surcharge de l'opérateur **+** dans le but de concaténer un unique caractère :

```
Chaine& operator+(char c)
{
  buffer[longueur++] = c;
  return *this;
}
```

Il est assez judicieux de renvoyer la **référence de l'objet courant**, ce qui permet d'enchaîner les opérations :

```
Chaine c("bonjour ");
c = c+'V';
c + 'o';
Chaine d;
d = c + 'u' + 's';  // enchaînement
d.afficher(); // affiche bonjour Vous
```

Vous aurez noté que notre opérateur + effectue une modification de l'instance à laquelle elle est appliquée, alors qu'en termes mathématiques additionner deux variables ne modifie ni l'une, ni l'autre. Il aurait été possible à la place de construire une nouvelle chaîne et de se contenter d'évaluer la concaténation. Nous gagnerions ainsi en souplesse mais perdrions vraisemblablement en performance.

Pratiquement tous les opérateurs sont disponibles pour la surcharge, à l'exception notable de ceux figurant dans le tableau suivant :

.	Accès aux champs
::	Résolution de portée
sizeof	Taille d'un type
? :	Forme prédicative (voir if)
.*	Adressage relatif des champsAdressage relatif

Ces exceptions faites, le programmeur a donc une grande liberté dans les sémantiques proposées par les opérateurs, à condition toutefois de conserver le nombre d'opérandes (binaire, unaire), la précédence, de respecter l'associativité. Le programmeur ne peut pas non plus déterminer l'adresse d'un opérateur (pointeur de fonction), ni proposer des valeurs par défaut pour les arguments applicables.

Les opérateurs peuvent être implémentés sous forme de méthode ou de fonctions, ces dernières ayant tout intérêt à être déclarées amies (friend). L'intérêt de cette approche est qu'il est possible d'inverser l'ordre des opérandes et d'augmenter ainsi le vocabulaire d'une classe existante.

Cette technique est d'ailleurs mise à profit pour la surcharge de l'opérateur **<<** applicable à la classe **std::ostream**.

b. Surcharge de l'opérateur d'indexation

Nous proposons son implémentation car il convient parfaitement à l'exemple de la classe **Chaine**. Cet opérateur est utile pour accéder à la nième donnée contenue dans un objet. Dans notre cas, il s'agit d'énumérer chaque caractère contenu dans une chaîne :

```
int length()
{
    return longueur;
}

char operator[](int index)
{
    return buffer[index];
}
```

L'affichage caractère par caractère devient alors trivial :

```
for(int i=0; i<d.length(); i++)
    printf("%c ",d[i]);
```

c. Surcharge de l'opérateur d'affectation

L'opérateur d'affectation est presque aussi important que le constructeur de copie. Il est fréquent de définir les deux pour une classe donnée.

```
Chaine& operator=(const Chaine& ch)
{
  if(this != &ch)
  {
    delete buffer;
    buffer = new char[ch.t_buf];   // ch référence
    for(int i=0; i<ch.longueur; i++)
      buffer[i] = ch.buffer[i];
    longueur = ch.longueur;
  }
  return *this;
}
```

d. Surcharge de l'opérateur de conversion

L'opérateur de conversion réalise un travail symétrique aux constructeurs prenant des types variés comme arguments pour initialiser le nouvel objet. Dans le cas de la classe **Chaine**, la conversion vers un **char*** est absolument nécessaire pour garantir une bonne traduction avec les chaînes gérées par le langage lui-même. Faites attention à la syntaxe qui est très particulière :

```
operator char*()
{
    char*out;
    out = new char[longueur+1];
    buffer[longueur] = 0;
    strcpy(out,buffer);
    return out;
}
```

L'application est beaucoup plus simple :

```
char*s=(char*) d;
printf("d=%s",s);
```

Cet opérateur sert à convertir par transtypage (cast) un objet de type **Chaine** vers le type **char***.

4. Fonctions amies

Les fonctions amies constituent un recours intéressant pour des fonctions n'appartenant pas à des classes mais devant accéder à leurs champs privés.

S'agissant de fonctions et non de méthodes, elles n'admettent pas de pointeur **this**, aussi reçoivent-elles généralement une instance de la classe comme paramètre, ou bien instancient-elles la classe en question.

```
class Chaine
{
...
  friend void afficher(Chaine&);
} ;

void afficher(Chaine& ch)
{
  for(int i=0; i<ch.longueur; i++)
```

```
    printf("%c",ch.buffer[i]);
}
```

Dans la déclaration de la classe **Chaine**, nous avons volontairement placé la déclaration d'amitié sans préciser s'il s'agissait d'un bloc public, privé ou protégé. Puisque la fonction **afficher()** n'est pas un membre, l'emplacement de sa déclaration n'a aucune importance.

Les fonctions amies présentent un réel intérêt pour surcharger des opérateurs dont le premier argument n'est pas la classe que l'on est en train de décrire. L'exemple habituel consiste à surcharger l'opérateur d'injection, **<<**, appliqué à la classe **ostream** :

```
#include <iostream>
using namespace std;

class Chaine
{
private:
  int t_buf;
  char*buffer;
  int longueur;
public:

  friend ostream& operator<<(ostream& out,Chaine&);
...
} ;

ostream& operator<<(ostream& out, Chaine& ch)
{
  for(int i=0; i<ch.longueur; i++)
    out << ch[i]; // << surchargé pour ostream et char

  return out;
}
int main(int argc, char* argv[])
{
  Chaine c("bonjour ");
  cout << c << "\n";  // cout est un objet de type ostream
}
```

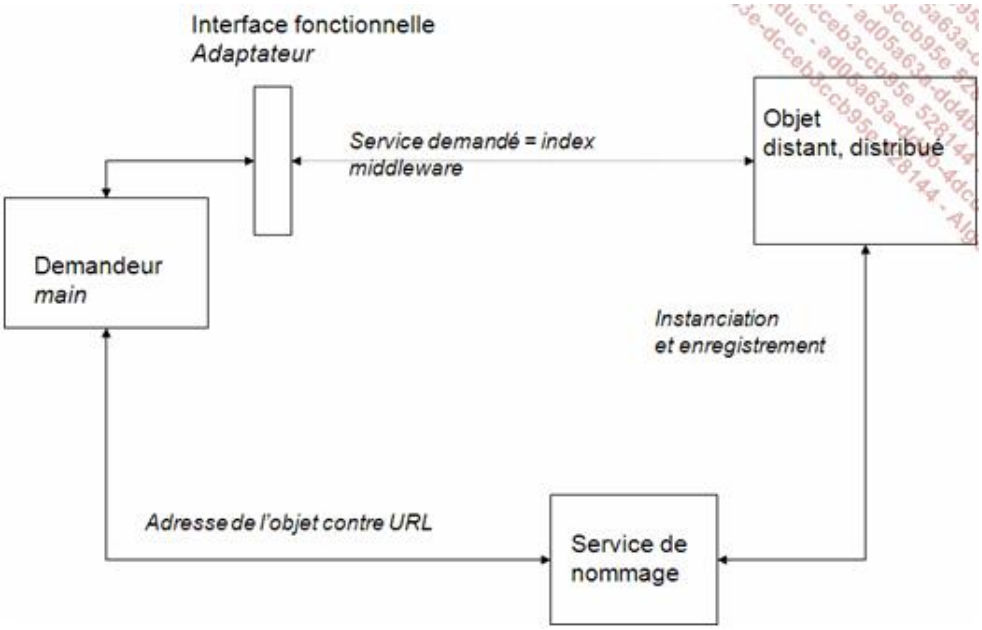
Signalons encore que les relations d'amitié ne sont pas héritables et qu'il est possible de définir des relations d'amitié entre méthodes. Ces usages sont toutefois à limiter pour augmenter la portabilité des programmes vers des langages ne connaissant pas ce concept.

5. Adressage relatif et pointeurs de membres

Le langage C++ a inauguré une nouvelle technique de programmation, la programmation dite distribuée. Cette technique consiste à appeler sur une machine distante des méthodes appliquées à un objet dont on ne connaît que l'interface. Rappelons-nous que l'interface équivaut à l'ensemble des méthodes publiques d'une classe.

Dans pareille situation, la notion de pointeur au sens où nous l'avons étudié jusqu'à présent est inapplicable. Un pointeur désigne une adresse appartenant à la machine courante, voire au processus courant. Pour invoquer une méthode, ou accéder à un champ à distance, un numéro de membre serait plus approprié, car il est indépendant de l'emplacement mémoire de l'objet. Bien entendu, au dernier moment, ce déplacement sera combiné à une adresse de base, physique celle-là, pour accéder au membre considéré.

Cette adresse de base n'est généralement pas connue du demandeur, il l'obtient à partir d'un service de nommage, souvent au moyen d'une URL appropriée.



De nombreux middlewares orientés objet, tels DCOM, Corba, RMI, ou .NET Remoting travaillent de cette façon. Beaucoup d'entre eux sont d'ailleurs implémentés en C++.

Les pointeurs de membres de C++ permettent justement d'exprimer l'accès à un membre en tant qu'index et non en tant qu'adresse physique.

a. Notations

Trois opérateurs sont dévoués à la prise en charge de l'adressage relatif, terme désignant en fait les pointeurs de membres :

- l'opérateur **&** (adresse relative d'un membre) ;
- l'opérateur **.*** (indirection sur une adresse relative) ;
- l'opérateur **->*** (indirection sur une adresse relative).

Le premier opérateur retrouve l'adresse relative d'un membre, en suivant la logique habituelle :

```
Chaine::* index_buffer=&Chaine::buffer;
```

Le type du pointeur est donc **`Classe::*`**.

Les autres opérateurs combinent l'index à une référence ou à une adresse d'objet, encore une fois en suivant la logique habituelle :

```
Chaine ch;
ch.*index_buffer=new char[100];
```

Pour les pointeurs relatifs de méthode, la signature est aussi conforme aux pointeurs de fonctions usuels :

```
int (Chaine::* pf_length)(void);
pf_length=&Chaine::length;
```

b. Construction d'un middleware orienté objet

Nous proposons d'appliquer les pointeurs de membres à la simulation d'un middleware servant à distribuer nos applications. Dans notre cas, tout fonctionnera à l'intérieur d'un seul processus mais le raisonnement est aisément généralisable à une infrastructure plus complexe.

Un middleware est un ensemble de services systèmes et réseau destiné à faire communiquer des logiciels fonctionnant sur des machines différentes. Il existe des middlewares orientés messages (Mom) dont MQSeries (IBM) et MSMQ sont les plus connus. Il s'agit ici d'étudier un middleware orienté objet (Moo).

Nous allons compléter le schéma présenté en introduction de cette partie à l'aide d'une classe Adaptateur. Il s'agit de la couche intermédiaire (middleware) chargée de relayer les invocations du demandeur à travers le "réseau" pour atteindre le véritable objet, désigné sous le nom d'objet distant.

Pour le demandeur, le scénario est le suivant :

1. accès au service de nommage (une seule instanciation).

2. localisation de l'objet distant contre la fourniture d'une URL.

3. accès aux méthodes distantes.

Pour illustrer notre exemple, nous proposons une interface **`IObjetDistant`** possédant deux méthodes, l'une se contentant de renvoyer la chaîne de caractères "Bonjour tout le monde" et la seconde, plus élaborée, additionnant deux nombres entiers transmis en paramètres.

Interface IObjetDistant

Voici pour commencer l'interface **`IObjetDistant`**, elle ne contient que des méthodes virtuelles pures :

```
#pragma once

// Fichier: IObjetDistant
// Propos : Cette interface contient la liste des méthodes
//          pouvant être appelées à distance.
//          Pour simplifier, toutes les méthodes ont la
//          même signature.
class IObjetDistant
{
public:
  virtual void* hello_world(int nb_params, void* params)=0;
  virtual void* somme(int nb_params, void* params)=0;
} ;
```

Implémentation ObjetDistant

Nous poursuivons avec son implémentation, qui ignore totalement qu'elle est amenée à être **`ObjetDistant`** distribuée, c'est-à-dire instanciée et invoquée à distance :

```
#pragma once

// Fichier: ObjetDistant.h

// Propos : Implémentation de la classe (interface) IObjetDistant
#include "IObjetDistant.h"

class ObjetDistant : public IObjetDistant
{
public:
  void* hello_world(int nb_params,void*params);
  void* somme(int nb_params,void*params);
```

```
};
```

Puis :

```
#include ".\objetdistant.h"

// la classe ObjetDistant ignore totalement qu'elle va être distribuée,
// c'est à dire appelée à distance.

void* ObjetDistant::hello_world(int nb_params,void*params)
{
  return "Bonjour tout le monde"; // char* en fait
}

void* ObjetDistant::somme(int nb_params,void*params)
{
  int*local_params=(int*) params;

  // pour renvoyer le résultat, un reinterpret_cast aurait été possible
  // mais peu rigoureux.
  int*resultat=new int[1];
  *resultat=local_params[0]+local_params[1];
  return (void*)resultat ;
}
```

Adaptateur

Nous entrons ensuite dans le vif du sujet avec la classe **Adaptateur**, qui utilise des pointeurs de membres à plusieurs niveaux.

Dans le cas d'une réelle distribution, il suffira de prévoir deux classes (**skeleton** et **stub**) translatant l'adresse de l'objet **l_objet** entre les deux machines.

```
#pragma once

// Fichier: Adaptateur.h
// Propos : Classe permettant l'invocation sécurisée de méthodes
//          sur l'objet distant.
#include "IObjetDistant.h"

class Adaptateur : public IObjetDistant
{
private:
  IObjetDistant*l_objet;

  void* invoquer_par_numero(
    void* (IObjetDistant::* p_fonction) (int,void*),
    int nb_params,void*params);
public:
  Adaptateur(IObjetDistant*adr);

  void* hello_world(int nb_params,void*params);
  void* somme(int nb_params,void*params);
};
```

L'implémentation suit naturellement. Notez comment les méthodes **hello_world()** et **somme()** se contentent d'appeler les véritables implémentations au moyen de pointeurs de membres :

```
#include ".\adaptateur.h"

Adaptateur::Adaptateur(IObjetDistant*adr)
{
  l_objet=adr;  // mémorise l'adresse physique de l'objet distant
}
```

```
void* Adaptateur::invoquer_par_numero(
  void* (IObjetDistant::* p_fonction) (int,void*) , int nb_params,void*params)
{
  // Le pointeur de membre est utilisé ici.
  // L'appel combine l'adresse physique, l_objet,
  // avec l'index de la fonction à appeler, p_fonction.
  return (l_objet->*p_fonction) (nb_params,params);
}

void* Adaptateur::hello_world(int nb_params,void*params)
{
  return invoquer_par_numero(
    &IObjetDistant::hello_world, // index de la fonction
    nb_params,params);
}

void* Adaptateur::somme(int nb_params,void*params)
{
  return invoquer_par_numero(
    &IObjetDistant::somme,    // index de la fonction
    nb_params,params);
}
```

Service de nommage

Le service de nommage est assez basique. Il assure ses responsabilités d'enregistrement d'objet (publication) avec celles d'un serveur d'objets (instanciation). Dans une situation réelle de distribution, l'URL désignant l'objet ne se résume pas à un seul segment indiquant le nom. Figure souvent aussi l'adresse du serveur où l'objet est instancié.

```
#pragma once

// Fichier: Nommage.h
// Propos : Système d'enregistrement et de localisation
// d'objets IObjetDistant
//          Pour simplifier, la classe publie automatiquement un objet
#include "IObjetDistant.h"
#include "Adaptateur.h"

class Nommage
{
private:
  IObjetDistant* objet;
  char*url;

public:
  void publier(IObjetDistant*objet,char*url);
  IObjetDistant*trouver(char*url);
  Nommage();
  ~Nommage();
};
```

L'implémentation est plus intéressante que la déclaration car elle substitue, à la demande du client, l'objet distant par sa version distribuée (Adaptateur) :

```
#include ".\nommage.h"
#include "objetdistant.h"
#include <string.h>

Nommage::Nommage()
{
  // procède à la publication d'un nouvel objet
  printf("Instanciation d'un objet sous le nom objet1\n");
  publier(new ObjetDistant,"objet1");
}
```

```
Nommage::~Nommage()
{
  // détruit l'objet
  delete this->objet;
  this->url=NULL;
}

void Nommage::publier(IObjetDistant*objet,char*url)
{
  printf("Publication d'un objet sous le nom %s\n",url);
  this->objet=objet;
  this->url=url;
}

IObjetDistant*Nommage::trouver(char*url)
{

  printf("Recherche d'un objet sous le nom %s\n",url);
  if(!strcmp(this->url,url))
    return new Adaptateur(this->objet);

  printf("Objet pas trouvé\n");
  return NULL;
}
```

Programme client (demandeur)

La fonction **`main()`** fait office du demandeur dans notre schéma. Elle est conforme au scénario évoqué ci-dessus :

```
#include "nommage.h"

int main(int argc, char* argv[])
{
  // Utilise le service de nommage ad-hoc
  Nommage nommage;

  // recherche d'un objet distant contre une url
  IObjetDistant*objet_distant= nommage.trouver("objet1");

  // appel d'une méthode à distance
  char*s=(char*) objet_distant->hello_world( 0,NULL);
  printf("s=%s\n",s);

  // appel d'une autre méthode, à distance toujours
  int tab[2]={3,8};
  int*r=(int*) objet_distant->somme(2,(void*) tab );
 printf("3+8=%d\n",*r);
  return 0;
}
```

Après tous ces efforts de programmation, nous proposons les résultats de l'exécution du programme :

6. La programmation générique

Le langage C est trop proche de la machine pour laisser entrevoir une réelle généricité. Au mieux, l'emploi de pointeurs **`void*`** (par ailleurs introduits par C++) permet le travail sur différents types. Mais cette imprécision se paye cher, tant du point de vue de la lisibilité des programmes, que de la robustesse ou des performances.

Les macros ne sont pas non plus une solution viable pour l'implémentation d'un algorithme indépendamment du typage. Les macros développent une philosophie inverse aux techniques de la compilation. Elles peuvent convenir dans certains cas simples mais leur utilisation relève le plus souvent du bricolage.

Reste que tous les algorithmes ne sont pas destinés à être implémentés pour l'universalité des types de données C++. Pour un certain nombre de types, on se met alors à penser en termes de fonctions polymorphes.

Finalement, les modèles C++ constituent une solution bien plus élégante, très simple et en même temps très sûre d'emploi. Le type de donnée est choisi par le programmeur qui va instancier son modèle à la demande. Le langage C++ propose des modèles de fonctions et des modèles de classes et nous allons traiter tour à tour ces deux aspects.

a. Modèles de fonctions

Tous les algorithmes ne se prêtent pas à la construction d'un modèle. Identifier un algorithme est une précaution simple à prendre car un modèle bâti à mauvais escient peut diminuer les qualités d'une implémentation.

Pour l'algorithmie standard, la bibliothèque S.T.L. a choisi d'être implémentée sous forme de modèles. Les conteneurs sont des structures dont le fonctionnement est bien entendu indépendant du type d'objet à mémoriser.

Pour les chaînes, le choix est plus discutable. La S.T.L. propose donc un modèle de classes pour des chaînes indépendantes du codage de caractère, ce qui autorise une ouverture vers des formats très variés. Pour le codage le plus courant, l'ASCII, la classe string est une implémentation spécifique (voir la partie sur les modèles spécialisés).

Le domaine numérique est particulièrement demandeur de modèles de fonctions et de classes. Le programmeur peut ainsi choisir entre précision (**`long double`** ou **`double`**) et rapidité (**`float`**) sans remettre en question l'ensemble de son programme. Certains algorithmes peuvent même travailler avec des représentations de nombres plus spécifiques.

Une syntaxe spécifique explicite la construction d'un modèle de fonction. Il s'agit d'un préfixe indiquant la liste des paramètres à fournir à l'instanciation du modèle. Les paramètres sont fréquemment écrits en majuscule pour les distinguer des variables dans le corps de la fonction, mais il ne s'agit aucunement d'une contrainte syntaxique.

```
template<class T> T cosinus(T a)
{
  int i;
  T cos = 1;
  T pa = a*a;
  int pf = 2;
  T signe = -1;

  for(i=2; i<=14; i+=2)
  {
    cos += signe*pa/pf;
    signe = -signe;
    pa = pa*a*a;
    pf = pf*(i+1);
    pf = pf*(i+2);
  }
  return cos;
}
```

Le modèle doit avant tout se lire comme une fonction dont le type est paramétrable. Imaginons que le type soit double, nous obtenons la signature suivante :

```
double cosinus(double a)
```

Dans le corps de la méthode, le type T s'écrit à la place du type des variables ou des expressions. Ainsi les variables **cos**, **pa** et **signe** pourraient être des **double** ou des **float**.

Il va sans dire que la liste des paramètres du modèle, spécifiée entre les symboles < et > peut être différente de la liste des paramètres de la fonction, ainsi que le prouve l'exemple suivant :

```
template<class T> void vecteur_add(T*vect,int n,T valeur)

{
  for(int i=0;i<n;i++)
    vect[i]+=valeur;
}
```

Pour utiliser un modèle de fonction, il faut simplement appeler la fonction avec des arguments qui correspondent à ceux qu'elle attend. Cela a pour effet de développer le modèle dans une version adaptée aux types des arguments transmis, arguments qui peuvent naturellement être des valeurs littérales, des expressions ou encore des variables, en respectant les règles habituelles d'appel de fonction.

Nous proposons un premier exemple avec notre modèle de fonction **cosinus**. Dans cet exemple, nous comparons la durée du calcul d'un nombre important de cosinus, ainsi que la précision du calcul. Suivant le type de l'argument passé à la fonction **cosinus** - **float** ou **long double** - le compilateur produira plusieurs versions à partir du même modèle.

```
int main(int argc, char* argv[])
{
  time_t   start, finish;
  double   result, elapsed_time;

  int N=30000000;
  // 30 000 000 cosinus en float
  time( &start );
  float a1=3.14159265358F/2;
  float r1;
  for(int i=0; i<N; i++)
    r1=cosinus(a1);
  time( &finish );
  elapsed_time = difftime( finish, start );

  printf("%d cosinus en float, r=%f\n",N,r1);
  printf("Durée du calcul :  %6.0f secondes.\n", elapsed_time );

  // 30 000 000 cosinus en long double
  time( &start );
  long double a2=3.14159265358/2;
  long double r2;
  for(i=0; i<N; i++)
    r2=cosinus(a2);
  time( &finish );
  elapsed_time = difftime( finish, start );

  printf("\n%d cosinus en long double, %g\n",N,r2);
  printf( "Durée du calcul :  %6.0f secondes.\n", elapsed_time );

  return 0;
}
```

Nous constatons des différences sensibles dans la durée et la qualité du calcul.

Munis de ces informations, le concepteur et le programmeur peuvent choisir le type de donnée le plus adapté à leur cahier des charges.

Nous proposons maintenant une application du modèle **vecteur_add()**. Un éditeur de code source performant guide le programmeur dans la saisie des arguments passés à la fonction :

Voilà maintenant le code complété :

```
double valeurs[]={3,-9,8.2,5 };
  vecteur_add(valeurs,4,3.0);
```

Le troisième paramètre de la fonction doit impérativement être du type correspondant au tableau de valeurs, dans notre cas pour des raisons d'homogénéité des calculs, mais aussi pour des raisons syntaxiques.

Fournir une valeur littérale ambiguë peut provoquer un avertissement, voire une erreur de compilation :

```
vecteur_add(valeurs,4,3);  // 3 littérale d'entier
```

Sachant que cet appel doit être le plus explicite possible, nous en déduisons que le modèle de fonction peut être surchargé, autrement dit, polymorphe.

La surcharge d'un modèle de fonction consiste à décrire une autre fonction portant le même nom mais recevant des paramètres différents.

```
template<class T> void vecteur_add(T& vect,T valeur)

{
  vect+=valeur;
}
```

Les paramètres du modèle peuvent également varier d'une version à l'autre.

À l'appel de la fonction, le compilateur décide quelle est la version la plus appropriée.

Nous obtenons finalement le code suivant pour instancier notre nouveau modèle :

```
vecteur_add(valeurs[0],(double) 3);
```

Lorsqu'une optimisation de l'implémentation est possible pour un type donné, il est d'usage de spécialiser le modèle. Ainsi, nous pourrions écrire une version spécifique de **cosinus()** s'appuyant sur la bibliothèque mathématique lorsque le type est double :

```
template<double> double cosinus(double a)
{
  return cos(a);
}
```

En présence du modèle général **template<class T> T cosinus(T a)** et de la version spécialisée, le

compilateur prendra la version spécialisée si le type considéré correspond exactement.

La classe string de la S.T.L. est un exemple de modèle spécialisé de classe **basic_string** appliqué aux **char**. Il va sans dire que les modèles de classes ont aussi la possibilité d'être spécialisés.

b. Modèles de classes

La spécification d'un modèle de classe suit exactement la même logique qu'un modèle de fonction. Nous donnons ici l'exemple d'une classe, **Matrice**, qui peut être implémentée avec différents types, au choix du programmeur.

```
template<class T> class Matrice
{
protected:
  int nb_vecteur,t_vecteur;
  T*valeurs;

public:
  // constructeur
  Matrice(int nb,int t) : nb_vecteur(nb),t_vecteur(t)
  {
    valeurs=new T[nb_vecteur*t_vecteur];
  }

  // référence sur le type T
  T& at(int vecteur,int valeur)
  {
    return valeurs[vecteur*t_vecteur+valeur];
  }

  // transtypage de double vers T
  void random()
  {
  int vect,val;
    for(vect=0; vect<nb_vecteur; vect++)
    {
      for(val=0; val<t_vecteur; val++)
        at(vect,val)=(T) (((double)rand()/RAND_MAX)*100.0);
    }
  }

  // affichage
  void afficher()
  {
    int vect,val;
    cout.precision(2);
    cout << fixed;
    for(vect=0; vect<nb_vecteur; vect++)
    {
      for(val=0; val<t_vecteur; val++)
      {
        cout.width(6);
        cout << at(vect,val) ;
        cout << (val<t_vecteur-1?",":"");
      }
      cout << endl;
    }
  }
} ;
```

Les membres de la classe **Matrice** illustrent différents aspects des modèles de classe.

La classe contient un champ de type **T***, comme nous avions pu déclarer précédemment une variable locale dans le corps de la fonction **cosinus**.

La méthode **at()** renvoie une référence vers un type **T**, noté **T&** bien entendu.

Le paramètre du modèle **T** peut être utilisé pour effectuer des conversions de type (transtypage) ; la méthode **random()** explicite le fonctionnement de la notation (**T**) pour initialiser de manière aléatoire chaque valeur de la matrice.

Quant à la fonction **afficher()**, elle est moins triviale qu'elle n'y paraît. Si nous devions utiliser **printf()** pour afficher sur la console nos valeurs, nous devrions tester le type réellement utilisé afin de choisir un formateur approprié, **%f** ou **%g** par exemple. Il se trouve que l'opérateur **<<** est surchargé pour les types primitifs de C++, mais pas **printf**.

Sans la classe **ostream**, nous aurions d'autres façons de résoudre ce problème ; remplacer **ostream**, ce qui peut se révéler fastidieux (et inutile). Nous pourrions également créer un modèle de fonction **afficher()**, externe à la classe, puis le spécialiser. Ou bien nous pourrions opérer une spécialisation partielle de la classe **Matrice**. Mais sur ce point, certains compilateurs se montrent plus coopératifs que d'autres. En attendant la solution indiquée au paragraphe consacré aux modèles de fonction, nous nous contentons de la version actuelle de la méthode **afficher()** qui fonctionne très bien avec les types usuels.

Si nous avions choisi une définition déportée de méthode, elle deviendrait un modèle de fonction. Il conviendrait alors de faire figurer les paramètres du modèle dans le nom de la classe :

```
template<class T> T& Matrice<T>::at(int vecteur,int valeur)
{
  return valeurs[vecteur*t_vecteur+valeur];
}
```

L'instanciation se passe presque comme pour une fonction, si ce n'est qu'il faut fournir explicitement le type associé au modèle.

```
Matrice<double> m(3,4);
m.random();
m.afficher();
```

Notre affichage produit les résultats suivants :

La syntaxe C++ admet des valeurs par défaut pour la définition de modèles :

```
template<class T=float> class Matrice
...
```

L'instanciation de la classe (et du modèle) peut alors se faire sans préciser de type, **float** étant la valeur par défaut :

```
Matrice<> mf(5,5);  // matrice de float
```

Nous en arrivons au sujet délicat de la spécialisation partielle. Précisons d'abord que les compilateurs C++ supportent plus ou moins bien ce mécanisme. Dans certains cas, la compilation n'aboutit pas ou bien le code généré est erroné, ce qui peut se révéler particulièrement gênant.

Parmi les compilateurs les plus réguliers, citons le GNU C++, le compilateur Microsoft VC++ et la dernière version fournie par la société Intel. D'autres sont évidemment conformes à 100 % à la norme C++ mais les produits cités ont fait la preuve de leur rigueur dans la version indiquée. Attention, car cela n'a pas toujours été

le cas, notamment pour le VC++ (avant 2003) et le compilateur d'Intel.

Nous allons commencer par créer un modèle de fonction pour la méthode **afficher()**, c'est-à-dire utiliser la définition déportée :

```
template<class T> void Matrice<T>::afficher()
{
  int vect,val;
  cout.precision(2);
  cout << fixed;
  for(vect=0; vect<nb_vecteur; vect++)
  {
    for(val=0; val<t_vecteur; val++)
    {
      cout.width(6);
      cout << at(vect,val) ;
      cout << (val<t_vecteur-1?",":"");
    }
    cout << endl;
  }
}
```

Cette version constitue la version de secours, la plus générale. Il se peut qu'elle soit impossible à construire, mais dans notre cas, **cout** se comporte assez bien avec l'ensemble des types usuels.

Nous poursuivons par la fourniture d'une version spécifique à l'affichage des **float**. On note que le modèle de fonction a perdu son paramètre mais que le type de classe, **Matrice<float>**, est indiqué :

```
template<> void Matrice<float>::afficher()
{
  int vect,val;

  cout.precision(2);
  cout << fixed;
  printf("Affichage spécifique float\n");
  for(vect=0; vect<nb_vecteur; vect++)
  {
    for(val=0; val<t_vecteur; val++)
    {
      printf("%f",at(vect,val)) ;
      printf((val<t_vecteur-1?", ":""));
    }
    printf("\n");;
  }
}
```

Comme pour les modèles de fonctions spécialisés, le compilateur utilisera de préférence cette version pour une matrice de **float** et la version la plus générale dans les autres cas.

Introduction

L'inventeur de C++, Bjarne Stroustrup, a travaillé sur des projets de développement très importants. Les premières applications de C++ ont été réalisées dans le domaine des télécommunications, domaine qui réclame des outils de haut niveau. Certes les classes favorisent l'abstraction, mais un programme n'est pas composé uniquement d'interfaces et d'implémentations.

À la conception du logiciel, le développeur doit déterminer la limite de réutilisation des réalisations précédentes. Derrière le terme réutilisation, on entend souvent le fait de copier/coller certaines parties de code source. Quels sont les éléments qui se prêtent le mieux à cette opération ? Les classes, naturellement, le modèle orienté objet étant construit autour du concept de réutilisation. Mais on trouve également des structures de données, des fonctions spécialisées, des pièces algorithmiques de natures diverses, n'étant pas encore parvenues à la maturité nécessaire à la formation de classes.

Les modules décrivent un découpage assez physique des programmes. Ainsi, un fichier de code source .h ou .cpp peut-il être considéré comme module. Toutefois, l'assemblage de modules n'est pas une chose aisée, surtout lorsqu'ils proviennent de réalisations précédentes. Il peut survenir des conflits de noms, de fonctions, des différences de représentation de types, ainsi qu'une déstructuration du programme, tous les membres d'un module apparaissant au même plan que les autres.

C'est pour ces raisons que C++ propose les espaces de noms ; il s'agit d'ensembles de variables, de fonctions, de classes et de sous-espaces de nom, membres obéissant à des règles de visibilité. Ainsi le programmeur pourra organiser et ordonner son développement, fruit d'un travail nouveau et d'une agrégation de codes sources anciens.

Langage de haut niveau, C++ incite aussi à prendre du recul sur les développements achevés. On ne peut que s'étonner que les mêmes algorithmes se retrouvent dans tous les logiciels. Un algorithme, c'est un élément caractérisé, au comportement déterminé. Pourquoi ne pas le formaliser à l'aide d'une ou de plusieurs classes ? C'est précisément un des rôles tenus par la bibliothèque standard, la Standard Template Library.

À dire vrai, la S.T.L. constitue tout aussi bien un ensemble d'outils pour le programmeur qu'une preuve du fonctionnement du modèle orienté objet.

Organisation des programmes

1. Espaces de noms

Le langage C ne connaît que deux niveaux de portée : le niveau global, auquel la fonction **main()** appartient, et le niveau local, destiné aux instructions et aux variables locales. Avec l'apparition de classes, un niveau supplémentaire s'est installé, celui destiné à l'enregistrement des champs et des méthodes. Puis l'introduction de la dérivation (héritage) et des membres statiques a encore nuancé la palette des niveaux de portée.

Pour les raisons évoquées en introduction, il devenait nécessaire de structurer l'espace global. Pour n'en retenir qu'une, l'espace global est trop risqué pour le rangement de variables et de fonctions provenant de programmes anciens. Les conflits sont inévitables.

On peut alors partitionner cet espace global à l'aide d'espaces de noms :

```
namespace Batiment
{
  double longueur;

  void mesurer()
  {
    longueur=50.3;
  }
} ;

namespace Chaines
{
  int longueur;

  void calcule_longueur(char*s)
  {
    longueur=strlen(s);
  }
} ;
```

Deux espaces de noms, **Batiment** et **Chaines**, contiennent tous les deux une variable nommée **longueur**, d'ailleurs de type différent. Les fonctions **mesurer()** et **calcule_longueur()** utilisent toujours la bonne version, car la règle d'accessibilité est également vérifiée dans les espaces de noms : le compilateur cherche toujours la version la plus proche.

Pour utiliser l'une ou l'autre de ces fonctions, la fonction **main()** doit recourir à l'opération de résolution de portée **::** ou bien à une instruction **using** :

```
int main(int argc, char* argv[])
{
  Batiment::mesurer();
  printf("La longueur du bâtiment est %g\n",Batiment::longueur);

  using Chaines::longueur;
  Chaines::calcule_longueur("bonjour");
  printf("La longueur de la chaîne est %d\n",longueur);
  return 0;
}
```

Nous remarquons que l'appel d'une fonction déclarée à l'intérieur d'un espace de noms ressemble beaucoup à celui d'une méthode statique. Cette analogie se prolonge pour l'accès à une variable, que l'on peut comparer à l'accès à un champ statique.

La syntaxe **using** est utile pour créer des alias. Avant la déclaration **using Chaines::longueur**, il n'existe pour **main()** aucune variable accessible. Ensuite, jusqu'à nouvel ordre, longueur est devenu un alias de **Chaines::longueur**.

a. Utilisation complète d'un espace de noms

Il n'est pas rare d'avoir à utiliser tous les membres appartenant à un espace de noms. Il est d'autant plus inutile de créer un alias pour chacun d'eux que l'opération risque d'être à la fois fastidieuse et vaine. La syntaxe **`using namespace`** convient bien mieux :

```
#include <iostream>
using namespace std;  // utilise tout l'espace de noms std
```

Cette déclaration est forte de signification, et il convient de bien la différencier de la directive **`#include`**. Alors que la directive **`#include`** inclut le fichier indiqué, **`iostream`** en l'espèce, la directive **`using namespace`** crée des alias pour les membres de l'espace de noms **`std`**, déclarés dans **`iostream`**.

Autrement dit, la directive **`using`** n'importe pas, n'inclut pas, elle est uniquement destinée à simplifier l'écriture. Sans son emploi, nous devrions préfixer chaque élément par **`std`**.

Ainsi est-il plus commode d'écrire :

```
cout << "Bonjour\n";
```

que :

```
std::cout << "Bonjour\n";
```

Toutefois, en cas de conflit entre deux espaces de noms entièrement utilisés, l'utilisation de l'opérateur de résolution de portée lève toute ambiguïté :

```
namespace Chaines
{
  int longueur;

  void calcule_longueur(char*s)
  {
    longueur=strlen(s);
  }

  bool cout;
} ;
...
  using namespace Chaines;
  std::cout << "Pas de doute sur cette ligne\n";
```

b. Espace de noms réparti sur plusieurs fichiers

Il est tout à fait juste de définir le même espace de noms au travers de différents fichiers de code source, chacun d'eux contribuant à l'enrichir de ses membres.

```
// tri.h

namespace Algo
{
  void tri_rapide(int*valeurs);
  void tri_bulle(int*valeurs);
} ;

// pile.h

namespace Algo
```

```
{
  class Pile
  {
  protected:
    int*valeurs;
    int nb_elements;
    int max;
  public:
    Pile();
    void empiler(int v);
    int depiler();
    bool pile_vide();
  } ;
} ;
...
#include "tri.h"
#include "pile.h"

using namespace Algo;

int main(int argc, char* argv[])
{
    return 0;
}
```

Le compilateur C++ de Microsoft, lorsqu'il fonctionne en mode managé (avec les extensions .NET), conserve l'organisation des modules au niveau du code objet et de l'exécutable. Du même coup, la différence entre la directive **#include** et **using namespace** devient de plus en plus ténue ; le langage C# a donc fait le choix de réunir les deux directives en une seule, **using**, un peu à la manière de Java avec l'instruction **import**.

D'autre part, il est tout à fait possible de conserver la séparation entre déclaration et définition. L'organisation du code source en **.h** et en **.cpp** fonctionne ainsi comme à l'accoutumée :

```
// tri.cpp
#include "tri.h"

void Algo::tri_rapide(int*valeurs)
{

}
```

c. Relation entre classe et espace de noms

Finalement, classe et espace de noms sont assez proches. Doit-on considérer l'espace de noms comme une classe ne contenant que des membres statiques ou la classe comme un module instanciable ? En réalité, une classe C++ engendre son propre espace de noms. Voilà qui éclaire à la fois l'utilisation de l'opérateur de résolution de portée et surtout, qui explique la déclaration des champs statiques.

Si le champ statique était effectivement "instancié" au moment de sa déclaration dans le corps de la classe, ce champ pourrait exister en plusieurs versions : chaque fichier **.cpp** incluant la déclaration de la classe au travers d'un fichier d'en-tête .h provoquant sa duplication. Il faut donc instancier ce champ, une et une seule fois, dans un fichier **.cpp**, normalement celui qui porte la définition des méthodes de la classe.

La mise en garde vaut aussi pour les espaces de noms. Si un espace est défini dans un fichier **.h**, il faut veiller à ne pas déclarer de variable dans cette partie, autrement la variable pourrait se trouver définie à plusieurs endroits.

Comment donc indiquer au lecteur l'existence d'une variable dans la déclaration **.h** d'un espace de noms, telle **cout** et **cin** pour l'espace **std**, sachant qu'elle ne peut pas être définie à cet endroit ? La réponse est encore le mot clé **static**. Le mot clé **static** prévient le compilateur que cette variable ne doit pas être dupliquée par chaque fichier incluant la définition de l'espace de noms.

Toutefois, et contrairement aux classes, la variable n'a pas à être définie dans un fichier **.cpp** :

```
// pile.h

namespace Algo
{
  class Pile
  {
  protected:
    int*valeurs;
    int nb_elements;
    int max;
  public:
    Pile();
    void empiler(int v);
    int depiler();
    bool pile_vide();
  } ;

  static int T_PILE=3;
} ;

// pile.cpp
#include "pile.h"

// ne pas déclarer ici la variable Algo::T_PILE
```

d. Déclaration de sous-espaces de noms

Un espace de noms peut très bien contenir d'autres espaces de noms, voire des classes et des structures.

```
namespace Algo
{
  namespace Structures
  {
    class Tableau {};
    class Pile {};
  } ;

  namespace Fonctions
  {
    using namespace Structures;
    void tri_rapide(Tableau t);
  } ;
} ;
```

Il est absolument nécessaire d'utiliser l'espace **`Structures`** dans l'espace **`Fonctions`** pour atteindre la définition de la classe **`Tableau`**. À l'extérieur de l'espace **`Algo`**, plusieurs directives peuvent être nécessaires pour utiliser l'ensemble des membres déclarés dans **`Algo`** :

```
using namespace Algo;
using namespace Algo::Structures;

int main(int argc, char* argv[])
{
  Tableau t;
  return 0;
}
```

2. Présentation de la STL

La bibliothèque standard a été créée dans le but d'aider le programmeur C++ à produire des logiciels de haut niveau. Bien que les langages C et C++ soient universellement supportés par des interfaces de programmation (API) de toutes sortes, la partie algorithmique reste un peu en retrait tant les instructions sont concises et

proches de la machine. Les chaînes de caractères ASCII-Z (terminées par un octet de valeur zéro) sont un bon exemple de situation pour laquelle le programmeur se trouve démuni. De nos jours, la question n'est plus de savoir comment les chaînes sont représentées mais plutôt comment les utiliser le mieux possible. Il faut conserver de bonnes performances, certes, mais il faut aussi s'accommoder des codages internationaux.

Aussi, la généralisation de la programmation orientée objet a fini par dénaturer le travail des développeurs. L'algorithmie a été abandonnée au profit d'une vision abstraite, à base d'interfaces. Bjarne Stroustrup avait peut-être senti que cette transformation du métier de programmeur interviendrait rapidement, aussi la S.T.L fût-elle rendue disponible très vite après la publication du langage.

La S.T.L. propose un certain nombre de thèmes pour aider le programmeur : les chaînes de caractères, les entrées-sorties, les algorithmes et leurs structures de données, le calcul numérique. Si cela ne suffisait pas à vos travaux, rappelez-vous qu'elle a été bâtie en C++, aussi tout programmeur est-il libre de la compléter avec ses propres contributions.

Flux C++ (entrées-sorties)

La S.T.L. gère de nombreux aspects des entrées-sorties. Elle inaugure une façon de programmer pour rendre persistants les nouveaux types définis à l'aide du langage C++.

L'équipe qui l'a conçue à la fin des années 80 a eu le souci d'être conforme aux techniques en vigueur et de produire un travail qui résisterait au fil des ans.

Il faut reconnaître que la gestion des fichiers a énormément évolué depuis l'introduction de la bibliothèque standard : les bases de données relationnelles ont remplacé les fichiers structurés, et les interfaces graphiques se sont imposées face aux consoles orientées caractères.

Toutefois, les axes pris pour le développement de la S.T.L. étaient les bons. Si l'utilisation des flux est un peu tombée en désuétude, leur étude permet d'y voir plus clair pour produire une nouvelle génération d'entrées-sorties. Aussi, le terminal en mode caractères continue son existence, la vitalité des systèmes Linux en est la preuve.

1. Flux C++

Pour bien commencer l'apprentissage des entrées-sorties, il faut faire la différence entre fichier et flux. Un fichier est caractérisé par un nom, un emplacement, des droits d'accès et parfois aussi un périphérique. Un flux - stream en anglais - est un contenu, une information qui est lue ou écrite par le programme. Cette information peut être de plus ou moins haut niveau. À la base, on trouve naturellement l'octet, puis celui-ci se spécialise en donnée de type entier, décimal, booléen, chaîne... Enfin, on peut créer des enregistrements composés d'informations très diverses. Il est tout à fait logique de considérer que la forme de ces enregistrements correspond à la formation d'une classe, c'est-à-dire d'un type au sens C++.

Les flux C++ - que l'on appelle parfois flots en langue française - sont organisés en trois niveaux ; le premier, le plus abstrait, regroupe les **ios_base**, format d'entrée-sortie indépendant de l'état et du formatage. Puis on trouve le niveau **basic_ios**, version intégrant la notion de paramètre régional (locale en anglais).

Enfin, nous trouvons le niveau **basic_iostream**, groupe de modèles de classes destinées à supporter le formatage de tous les types de base connus par le langage. C'est à ce niveau que nous travaillerons.

La circulation des informations se fait dans le cadre d'un système de mémoire tampon (buffer), supporté par la classe **basic_streambuf**.

Le programmeur-utilisateur de la S.T.L. connaît généralement les flux standard, **cout**, **cin** et **cerr**, ainsi que les classes **istream** et **ostream**.

2. Flux intégrés

Les flux intégrés, **cout**, **cin** et **cerr**, offrent de nombreuses façons d'échanger des informations. Les objets **cout** et **cerr** sont des instances de la classe **ostream** tandis que **cin** est une instance de la classe **istream**.

Ces classes proposent les opérateurs **<<** et **>>** pour lire ou écrire les types primitifs :

```
cout << "Bonjour"; // char*
int age;
cin >> age;
```

Nous avons vu qu'il était tout à fait possible d'enrichir le registre des types supportés en surchargeant cet opérateur à l'aide d'une fonction amie :

```
friend ostream& operator << (ostream& out,Type & t);
friend istream& operator >> (istream& out,Type & t);
```

3. État d'un flux

Les flux **istream** et **ostream** disposent de méthodes pour caractériser leur état.

setstate(iostate)	Ajoute un indicateur d'état
clear(iostate)	Définit les indicateurs d'état

4. Mise en forme

La classe **ios_base** propose un certain nombre de contrôles de formatage applicables par la suite. Ces contrôles s'utilisent comme des interrupteurs. On applique un formatage qui reste actif jusqu'à nouvel ordre.

skipws	Saute l'espace dans l'entrée
left	Ajuste le champ en le remplissant après la valeur
right	Remplit avant la valeur
internal	Remplit entre le signe et la valeur
boolalpha	Utilise une représentation symbolique pour true et false
dec	Base décimale
hex	Base hexadécimale
oct	Base octale
scientific	Notation avec virgule flottante
fixed	Format à virgule fixe dddd.dd
showbase	Ajoute un préfixe indiquant la base
showpoint	Imprime les zéros à droite
showpos	Indique + pour les nombres positifs
uppercase	Affiche E plutôt que e
adjustfield	Lié à l'ajustement du champ : internal, left ou right
basefield	Lié à la base : dec, hex ou oct
floatfield	Lié à la sortie en flottant : fixed ou scientific
flags()	Lit les indicateurs
flags(fmtflags)	Définit les indicateurs
setf(fmtflags)	Ajoute un indicateur
unsetf(fmtflags)	Annule un indicateur

À l'aide des formateurs, nous pouvons afficher temporairement en hexadécimal une valeur entière :

```
cout << hex << 4096 << "\n";
cout << 8192 << "\n";     // toujours en hexa
cout << dec << 4096 << "\n"; // repasse en décimal
```

La fonction **`width()`** peut spécifier l'espace destiné à la prochaine opération d'entrée-sortie, ce qui est utile pour l'affichage de nombres. La méthode **`fill()`** fournit le caractère de remplissage :

```
cout.width(8);
cout.fill('#');
cout << 3.1415;
```

La S.T.L. propose également un certain nombre de manipulateurs de flux, comme **`flush`**, qui assure la purge du flux, c'est-à-dire le vidage du tampon à destination du périphérique de sortie :

```
cout << 3.1415 << flush;
```

5. Flux de fichiers

Pour travailler avec les fichiers d'une autre manière que le ferait le langage C, la S.T.L. propose les classes **`ofstream`** et **`ifstream`**. Il s'agit, pour les fichiers, d'adaptation par héritage des classes **`ostream`** et **`istream`**.

Voici pour commencer un programme de copie de fichier utilisant ces classes :

```
#include <fstream>

using namespace std;

int main(int argc, char* argv[])
{
  if(argc<3)
    return 1;

  // ouverture des flux
  ifstream src(argv[1]);
  if(!src)
  {
    cerr << "Impossible d'ouvrir " << argv[1] << endl;
    return 2;
  }

  ofstream dst(argv[2]);
  if(!dst)
  {
    cerr << "Impossible d'ouvrir " << argv[2] << endl;
    return 2;
  }


  // copie
  char c;
  while(src.get(c))
    dst.put(c);

  // fermeture
  src.close();
  dst.close();

  return 0;
}
```

Les classes possèdent la même logique que leurs ancêtres **ostream** et **istream**. La boucle de copie aurait donc pu être écrite de la manière suivante :

```
// copie
  char c;
  while(! src.eof())
  {
    src >> c;
    dst << c;
  }
```

Il existe une différence importante avec l'API proposée par le langage C : les quantités numériques sont traitées comme des chaînes de caractères. L'ouverture en mode binaire ne changerait rien à ce comportement, **ofstream** étant spécialisé dans le traitement des caractères (char).

```
sortie.open("fichier.bin",ios_base::binary);
sortie << x;  // toujours la chaîne "43"
sortie.close();
```

6. Flux de chaînes

Un flux peut être attaché à une chaîne de caractères (**string**) plutôt qu'à un périphérique. Cette opération rend le programme plus générique et permet d'utiliser les séquences de formatage pour des messages avant une impression dans un journal (log), un fichier, un écran...

Pour utiliser les flux de chaînes, il faut inclure le fichier **<sstream>**, ainsi que l'expose le programme suivant :

```
#include <math.h>
#include <sstream>
#include <iostream>

using namespace std;

int main(int argc, char* argv[])
{
  ostringstream out;
  double pi = 3.14159265358;
  out.width(8);
  out.setf(ios_base::scientific);
  out << "cos(pi/4)= " << cos(pi/4) << endl;

  // sortie du flux sur une string
  string res = out.str();

  // finalement affichage
  cout << res;

  // relecture
  istringstream in(res);
  double x;
  string msg;
  in >> msg >> x;
  cout << "msg=" << msg << endl << "x=" << x;
  return 0;
}
```

L'exécution de ce programme est assez intéressante. À la relecture, le scanner stoppe la chaîne **msg** sur le caractère espace inséré après l'espace. D'autre part, nous retrouvons bien la valeur de la variable x, soit racine de 2 sur 2 (0.707) :

7. Les paramètres locaux

La classe locale définit un certain nombre de facettes destinées à être prises en compte lors du formatage par les flux de la S.T.L. Chaque facette se charge d'un type de formatage pour une culture particulière. Imaginons pour l'exemple le formatage d'un nombre en monnaie pour un pays de la zone Euro, ou bien le formatage de la date en langue allemande.

Cette classe sert à choisir un certain nombre de facettes en vue de leur application sur un flux par l'intermédiaire de la méthode **`imbue()`**. Elle contient aussi un certain nombre de méthodes statiques pour comparer deux classes locales entre elles.

Pour illustrer le fonctionnement subtil des classes locales définies par la S.T.L., nous allons concevoir une facette pour afficher des nombres en Euro. L'exemple pourra ensuite être aménagé pour supporter différents symboles monétaires.

Pour commencer, nous allons inclure les fichiers correspondant à l'emploi de classe locale, ainsi que l'en-tête **`fstream`**, car notre système utilise le flux **`cout`** pour l'affichage :

```
#include <fstream>
#include <locale>
using namespace std;
```

À présent, nous définissons une structure (classe) **`Monnaie`** pour que **`cout`** fasse la distinction entre l'affichage d'un **`double`** - sans unité - et l'affichage d'une donnée de type **`Monnaie`** :

```
struct Monnaie
{
    Monnaie (const double montant)
        : m(montant) {}

    double m;
};
```

Vous aurez sans doute remarqué le style d'initialisation de la variable m, très courant en C++ (c'est comme si l'on utilisait le constructeur de la variable m).

Nous poursuivons par l'écriture d'une facette destinée à formater la donnée pour l'affichage. Il est facile de sous-classer cette facette en vue de la rendre paramétrable.

```
class monnaie_put: public locale::facet
{
public:
  static locale::id id;

  monnaie_put (std::size_t refs = 0)
    : std::locale::facet (refs) { }

  string put (const double &m) const
  {
    char buf[50];
    sprintf(buf,"%g Euro",m);
    return string(buf);
```

```
  }
};

locale::id monnaie_put::id;
```

Suivant l'organisation de votre programme, la définition du champ **monnaie_put::id** devra se faire dans un fichier **.cpp** séparé.

Nous allons maintenant surcharger l'opérateur **<<** pour supporter l'insertion d'un objet **Monnaie** dans un flux **ostream**. S'agissant d'une structure dont tous les champs sont publics, la déclaration d'amitié n'est pas nécessaire.

Nous obtenons directement :

```
ostream& operator<< (ostream& os, const Monnaie& mon)
{
  std::locale loc = os.getloc ();
  const monnaie_put& ppFacet
                    = std::use_facet<monnaie_put> (loc);
  os << ppFacet.put(mon.m);
  return (os);
}
```

Il ne nous reste plus qu'à appliquer cette construction et à tester le programme.

```
int main(int argc, char* argv[])
{
  cout.imbue (locale (locale::classic (), new monnaie_put));
  Monnaie m(30);
  cout << m;  // affiche 30 Euro
  return 0;
}
```

Classe string pour la représentation des chaînes de caractères

C'est un fait étonnant, la majorité des traités d'algorithmie n'étudie pas les chaînes en tant que telles. La structure de données s'en rapprochant le plus reste le tableau pour lequel on a imaginé une grande quantité de problèmes et de solutions.

Le langage C est resté fidèle à cette approche et considère les chaînes comme des tableaux de caractères. Ses concepteurs ont fait deux choix importants ; la longueur d'une chaîne est limitée à celle allouée pour le tableau, et le codage est celui des caractères du C, utilisant la table ASCII. Dans la mesure où il n'existe pas de moyen de déterminer la taille d'un tableau autrement qu'en utilisant une variable supplémentaire, les concepteurs du langage C ont imaginé de terminer leurs chaînes par un caractère spécial, de valeur nulle. Il est vrai que ce caractère n'a pas de fonction dans la table ASCII, mais les chaînes du C sont devenues très spécialisées, donc, très loin de l'algorithmie générale.

L'auteur de C++, Bjarne Stroustrup, a souhaité pour son langage une compatibilité avec le langage C mais aussi une amélioration du codage prenant en compte différents formats de codage, ASCII ou non.

1. Représentation des chaînes dans la S.T.L

Pour la S.T.L., une chaîne est un ensemble ordonné de caractères. Une chaîne s'apparente donc fortement au **`vector`**, classe également présente dans la bibliothèque. Toutefois, la chaîne développe des accès et des traitements qui lui sont propres, soutenant ainsi mieux les algorithmes traduits en C++.

Les chaînes de la bibliothèque standard utilisent une classe de caractères pour s'affranchir du codage. La S.T.L. fournit le support pour les caractères ASCII (**`char`**) et pour les caractères étendus (**`wchar_t`**), mais on pourrait très bien envisager de développer d'autres formats destinés à des algorithmes à base de chaînes. Le génie génétique emploie des chaînes composées de caractères spécifiques, A, C, G, T. Le codage avec un **`char`** est donc très coûteux en terme d'espace, puisque deux bits suffisent à exprimer un tel vocabulaire, d'autant plus que les séquences de gènes peuvent concerner plusieurs centaines de milliers de bases. On peut aussi spécifier des caractères adaptés à des alphabets non latins, pour lesquels la table ASCII est inefficace.

La classe **`basic_string`** utilise un vecteur (**`vector`**) pour ranger les caractères en mémoire. L'implémentation ainsi que les performances peuvent varier d'un environnement à l'autre, suivant la qualité de la programmation. Toutefois, la conséquence la plus intéressante de l'utilisation du vecteur est que la longueur des chaînes est devenue variable. Voilà une limitation du langage C enfin dépassée.

En contrepartie, l'accès aux caractères n'est pas aussi direct. Des méthodes spécifiques ont été ajoutées, la classe **`vector`** ne proposant pas le nécessaire. Une chaîne propose également des méthodes spécifiques pour travailler sur des intervalles de caractères (extraction de sous-chaînes, recherche), alors que le vecteur est plutôt destiné à l'accès individuel aux éléments qu'il contient.

Pour le programmeur-utilisateur de la bibliothèque standard, la partie chaîne de caractères se résume peut-être à la classe **`string`**, destinée à améliorer l'antique **`char*`**. Mais il y a aussi le matériel nécessaire pour aller plus loin.

2. Mode d'emploi de la classe string

La classe **`string`** est une spécialisation du modèle **`basic_string`** pour les caractères habituels, **`char`**. Les caractéristiques exposées par la suite sont également valides pour d'autres formats de chaînes issus de **`basic_string`**.

a. Fonctions de base

Une chaîne se construit de différentes manières, à partir d'une littérale de chaîne ou caractère par caractère. Lorsque la chaîne est créée, on cherche souvent à accéder à certains de ses caractères.

Constructeurs

Différents constructeurs sont disponibles pour initialiser une chaîne de type string. Une chaîne peut être initialisée à partir d'une chaîne C (**char***), à partir d'une autre chaîne ou d'un morceau de chaîne :

```
#include <iostream>
#include <string>
using namespace std;

int main(int argc, char* argv[])
{
  string s1;  // chaîne vide
  string s2 = ""; // vide aussi

  string s3 = "Bonjour";
  string s4(s3);  // copie s3 dans s4

  // copie tous les caractères de s3 dans s5
  string s5(s3.begin(),s3.end());
  string s6(s3,0,s3.length());  // idem
  cout << s1 << " " << s2 << endl;
  cout << s3 << " " << s4 << endl;
  cout << s5 << " " << s6 << endl;

  return 0;
}
```

Les méthodes **begin()** et **end()** expriment des positions de sous-chaîne. Il ne s'agit pas de valeurs entières, mais de références à des caractères. Le constructeur utilisé pour **s6** est donc différent de celui employé pour **s5** qui fonctionne avec des indices de caractères.

Itérateurs

La classe string fournit deux itérateurs destinés à l'itération ordinaire et à l'itération inverse. Toutefois, les méthodes spécifiques de la classe **string** donnent de meilleures implémentations pour les algorithmes spécifiques aux chaînes.

Il est cependant possible d'employer ces itérateurs, **begin/end** (respectivement **rbegin**, **rend**) :

```
#include <algorithm>
#include <iostream>
#include <string>
using namespace std;

int main(int argc, char* argv[])
{
  string s="Bonjour la STL";
  string::iterator p=find(s.begin(),s.end(),'n');
  if(p==s.end())
    cout << "Il n'existe pas de caractère 'n' dans s" << endl;
  else
    cout << "Le caractère 'n' se trouve dans la chaîne : " << *p;
  return 0;
}
```

Accès aux éléments

L'opérateur d'index **[]** a été surchargé pour la classe **string**. On accède alors aux éléments **s[0]** à **s[s.length() -1]**. En dehors de ces plages, l'accès déclenche une exception **out_of_range**.

```
string s="Bonjour la STL";
cout << s[0] << ' ' << s[1] << ' ' <<
s[s.length()-1] << endl;
```

Attention car l'équivalence entre tableau et pointeur n'est plus vérifiée, la classe **string** ayant défini plusieurs champs. Ainsi, s est différent de **&s[0]**.

b. Intégration dans le langage C++

La chaîne est un type singulier pour tout langage de programmation, aussi important que peut l'être l'entier ou le booléen. La classe **string** a été conçue dans le but de rendre son emploi le plus naturel possible, c'est-à-dire qu'elle doit s'intégrer aussi bien que **char*** au sein des programmes C++.

Affectations

Nous avons vu au chapitre sur la Programmation orientée objet qu'affectation et constructeur sont des notions liées. La classe **string** possède évidemment un constructeur de copie, dont trois arguments sur quatre reçoivent des valeurs par défaut.

L'opérateur = a été redéfini pour rendre l'emploi de la classe le moins singulier possible :

```
string m;
  m = 'a';    // initialiser m à partir de 'a'
  m = "Bonjour"; // initialise m à partir de char*
  m = s;
```

Attention toutefois de ne pas commettre d'erreur de sémantique liée à la confusion entre table ASCII et valeur entière :

```
m = 65L;    // erreur de sémantique
```

En effet, m serait initialisée avec le caractère de code ASCII 65, donc 'A'. et non avec la chaîne "65".

Comparaisons

La comparaison de chaînes est essentielle pour implémenter les algorithmes. La classe **basic_string** supporte les comparaisons entre deux chaînes et entre une chaîne et un tableau de caractères. La méthode **compare()** retourne -1, 0 ou 1 comme le ferait la fonction **strcmp()**.

De plus, les opérateurs **==**, **!=**, **>**, **<**, **<=** et **>=** ont été surchargés pour comparer des chaînes.

```
#include <iostream>
#include <string>
using namespace std;

int main(int argc, char* argv[])
{
  string a="Piano";
  string b="Pianissimo";

  cout << boolalpha;
  cout << a.compare(b) << endl;    // affiche 1
  cout << a.compare("Forte") << endl;  // affiche 1

  cout << (a<b) << endl;        // affiche true

  return 0;
}
```

Conversion avec le C

Il existe trois méthodes pour appliquer les fonctions de traitement des chaînes du C sur des instances de la classe **string : data()**, **c_str()** et **copy()**.

La méthode **data()** copie les caractères de la chaîne dans un tableau avant de retourner un pointeur vers cette zone (**const char***). L'instance de la classe **string** a la charge de libérer de la mémoire le tableau, aussi le programmeur ne doit pas tenter de le faire lui-même. Si la chaîne est modifiée pendant la période d'exposition de la fonction **data()**, des caractères risquent d'être perdus :

```
string a="bonjour";
const char*d=a.data();
a[0]='B';
char c=d[0];  // erreur, toujours la valeur 'b'
```

La méthode **c_str()** ajoute un caractère **0** à la fin de la chaîne, faisant coïncider cette représentation avec les chaînes du langage C :

```
const char*b = a.c_str();
int l_a = strlen(b);
```

Enfin, la méthode **copy()** recopie le contenu de la chaîne dans un tampon dont le programmeur a la responsabilité de libération :

```
char*t=new char[a.length()+1];  // 0 terminal
  a.copy(t,a.length());
  t[a.length()]=0;  // 0 terminal
  // ...
  delete t;
```

c. Fonctions spécifiques aux chaînes

Les chaînes de caractères de la bibliothèque standard offrent des méthodes spécifiques à l'écriture de certains algorithmes. Il s'agit de fonctions de haut niveau, difficilement programmables en langage C pour certaines d'entre elles.

Insertion

L'opérateur **+=** a été surchargé pour l'ajout d'un caractère ou d'une chaîne à la fin d'une chaîne. Cet opérateur a la même fonction que la méthode **append()** qui offre elle un registre de possibilités plus varié.

La méthode **insert()** présente l'avantage d'intercaler de nouveaux caractères à l'endroit souhaité par le programmeur.

```
string s="Fort";
  s+="isimo";
  cout << "s=" << s << endl;  // affiche Fortisimo
  s.insert(5,"s");
  cout << "s=" << s << endl;  // affiche Fortissimo
```

Concaténation

L'opérateur **+** a été surchargé pour réunir plusieurs chaînes en une seule. Il concatène aussi bien des caractères que des chaînes :

```
string tonalite;
string ton="la ";
ton=ton+'b';
string maj="majeur";
tonalite=ton+maj;
```

```
cout << tonalite << endl; // affiche la b majeur
```

Recherche et remplacement

Il existe plusieurs méthodes pour rechercher des éléments à l'intérieur d'une chaîne :

find	recherche d'une sous-chaîne.
rfind	recherche en partant de la fin.
find_first_of	recherche de caractères.
find_last_of	recherche de caractères en partant de la fin.
find_first_not_of	recherche d'un caractère absent dans l'argument.
find_last_not_of	recherche d'un caractère absent dans l'argument en partant de la fin.

Toutes ces méthodes renvoient un **string::size_type**, assimilable à un entier.

```
string s="BEADGCF";
string::size_type i1=s.find("A");

cout << "i1=" << i1 << endl;  // affiche 2
```

Si la recherche n'aboutit pas, **find()** renvoie **npos**, position de caractère illégale. La méthode **find()** renvoie d'ailleurs un résultat non signé :

```
string::size_type i2=s.find("Z");
cout << "i2=" << i2 << endl;  // affiche un nombre très
grand
if(i2==string::npos)
  cout << "recherche de Z non trouvé";
```

Il n'est pas rare de combiner recherche et remplacement. C'est précisément le rôle de la méthode **replace()** que de modifier une séquence de chaîne en la remplaçant par une autre séquence :

```
string ton="la bémol majeur";
string::size_type p=ton.find("maj");
ton.replace(p,6,"mineur");
cout << ton << endl;    // affiche la bémol mineur
```

Il faut remarquer que la séquence remplaçante peut avoir une longueur différente de la séquence recouverte. Il existe aussi une méthode **erase()** pour supprimer un certain nombre de caractères :

```
ton.erase(ton.find("bémol"),5);
cout << ton << endl;    // affiche la mineur
```

Extraction de sous-chaîne

La méthode **substr()** réalise une extraction d'une sous-chaîne. Il est entendu qu'elle renvoie un résultat de type **string**. Voici un exemple très simple d'utilisation de cette méthode :

```
cout << ton.substr(0,3).c_str() << endl; // affiche la
```

Conteneurs dynamiques

Une fonction essentielle de la bibliothèque standard est de fournir des mécanismes pour supporter les algorithmes avec la meilleure efficacité possible. Cet énoncé comporte plusieurs objectifs contradictoires. Les algorithmes réclament de la généricité, autrement dit des méthodes de travail indépendantes du type de données à manipuler. Le langage C utilisait volontiers les pointeurs **`void*`** pour garantir la généricité mais cette approche entraîne une perte d'efficacité importante dans le contrôle des types, une complication du code et finalement de piètres performances. L'efficacité réclamée pour S.T.L.. s'obtient au prix d'une conception rigoureuse et de contrôles subtils des types. Certes, le résultat est un compromis entre des attentes parfois opposées mais il est assez probant pour être utilisé à l'élaboration d'applications dont la sûreté de fonctionnement est impérative.

Les concepteurs de la S.T.L. ont utilisé les modèles de classes pour développer la généricité. Les modèles de classes et de fonctions sont étudiés en détail au chapitre Programmation orientée objet, et leur emploi est assez simple. Une classe est instanciée à partir de son modèle en fournissant les paramètres attendus, généralement le type de données effectivement pris en compte pour l'implémentation de la classe. Il faut absolument différencier cette approche de l'utilisation de macros (**`#define`**) qui provoque des résultats inattendus. Ces macros se révèlent très peu sûres d'emploi.

Une idée originale dans la construction de la bibliothèque standard est la corrélation entre les conteneurs de données et les algorithmes s'appliquant à ces conteneurs. Une lecture comparée de différents manuels d'algorithmie débouche sur la conclusion que les structures de données sont toujours un peu les mêmes, ainsi que les algorithmes s'appliquant à ces structures. Il n'était donc pas opportun de concevoir des structures isolées, comme une pile ou une liste sans penser à l'après, à l'application d'algorithmes moins spécifiques. Les concepteurs de la S.T.L. ont su éviter cet écueil.

1. Conteneurs

Les conteneurs sont des structures de données pour ranger des objets de type varié. Les conteneurs de la S.T.L. respectent les principales constructions élaborées en algorithmie.

Classe	Description	En-tête
`vector`	Tableau de T à une dimension. Structure de référence.	**`<vector>`**
`list`	Liste doublement chaînée de T.	**`<list>`**
`deque`	File d'attente (queue en anglais) de T à double accès.	**`<deque>`**
`priority_queue`	File d'attente à priorité.	**`<deque>`**
`stack`	Pile de T.	**`<stack>`**
`map`	Tableau associatif de T.	**`<map>`**
`multimap`	Tableau associatif de T.	**`<map>`**
`set`	Ensemble de T.	**`<set>`**
`multiset`	Ensemble de T.	**`<set>`**
`bitset`	Tableau de booléens, ensemble de bits.	**`<bitset>`**

La classe **`vector`** est un peu une structure de référence. Bien que les autres classes ne soient pas basées sur celle-ci pour des raisons de performances, **`vector`** ouvre la voie aux autres conteneurs. Cette première partie explicite son fonctionnement, tandis que la partie suivante est consacrée aux séquences, c'est-à-dire aux conteneurs de plus haut niveau, mais reprenant la logique du vecteur.

a. Insertion d'éléments et parcours

Les deux principales fonctions attendues des conteneurs sont de pouvoir agréger de nouveaux éléments et de parcourir les éléments référencés par le conteneur.

L'insertion est une opération dont le déroulement est généralement propre au type de conteneur considéré. Par exemple, l'insertion dans une file d'attente à priorité est sensiblement différente de l'insertion dans un tableau associatif.

Le parcours, lorsqu'il est complet, est fréquemment impliqué dans l'inspection de la collection d'objets que représente le conteneur. Ce parcours est également envisageable pour explorer les résultats fournis par un algorithme de sélection ou de recherche.

b. Itérateurs

Lorsque le programmeur effectue une recherche, un tri, une sélection ou d'autres opérations de ce type sur un conteneur, il n'a pas besoin de rendre son programme dépendant du conteneur utilisé. En effet, le parcours d'une sélection sur un tableau associatif ou sur un vecteur est identique d'un point de vue algorithmique. Les itérateurs ont justement pour mission de fournir un moyen général pour parcourir les données.

Voici maintenant un exemple de construction d'un vecteur de chaînes et de parcours à l'aide d'un itérateur :

```
#include <iostream>
#include <string>
#include <vector>
#include <iterator>

using namespace std;

int main(int argc, char* argv[])
{
  vector<string> tab;
  vector<string>::iterator iter;

  tab.insert(tab.begin(),"Mozart");
  tab.insert(tab.begin()+1,"Beethoven");
  tab.insert(tab.begin()+2,"Schubert");

  for(iter=tab.begin(); iter!=tab.end(); iter++)
    cout << (*iter) << " ";

  cout << endl;
  return 0;
}
```

Un itérateur s'utilise donc à la manière d'un pointeur dont l'arithmétique est propre au conteneur considéré.

Les itérateurs servent au parcours des éléments d'un conteneur indépendamment du type réel de conteneur. Chaque conteneur possède des fonctions membres pour déterminer les bornes (début et fin) de la séquence d'éléments.

c. Opérations applicables à un vecteur

Le vecteur est un tableau dynamique, une collection d'éléments du même type. Le tableau peut également être considéré comme une pile, structure très importante pour l'écriture de programmes. Les méthodes **push_back()**, **pop_back()** et **back()** sont justement destinées à accomplir ce rôle.

```
vector<int> v;
  v.push_back(1756);  // empile 1756
  v.push_back(1770);  // empile 1770
  v.push_back(1797);  // empile 1797

  int last=v.back();  // interroge le dernier élément, soit 1797
```

```
  v.pop_back();    // dépile 1797

  vector<int>::iterator i;
  for(i=v.begin(); i!=v.end(); i++)
    cout << *i << " ";
```

Le vecteur possède également un certain nombre de méthodes pour l'insertion et l'effacement de données à des emplacements précis alors que les fonctions de piles agissent à l'extrémité de la séquence.

```
vector<string> opus;
  opus.insert(opus.begin(),"K545");
  opus.insert(opus.begin()+1,"K331");
  opus.erase(opus.begin());
  vector<string>::iterator o;

  for(o = opus.begin(); o != opus.end(); o++)
    cout << *o << " "; //affiche seulement K331
```

Enfin le vecteur possède une fonction **swap()** qui échange les valeurs entre deux vecteurs, opération très utile pour le tri des éléments.

2. Séquences

a. Conteneurs standard

La plupart des conteneurs suivent le modèle de **vector**, sauf lorsque cet "héritage" dénature le rôle accompli par le conteneur. Nous pouvons d'ores et déjà dégager un certain nombre de caractéristiques et d'opérations communes.

Types de membres

value_type	Type d'élément
allocator_type	Type du gestionnaire d'élément
size_type	Type des indices, des comptes d'éléments
difference_type	Type de différence entre deux itérateurs
iterator	espèce de value_type*
const_iterator	type de const value_type*
reverse_iterator	itérateur en ordre inverse, value_type*
const_reverse_iterator	const value_type*
reference	value_type&
const_reference	const value_type&

Itérateurs

begin()	Pointe sur le premier élément
end()	Pointe sur l'élément suivant le dernier élément

rbegin()	Pointe sur le dernier élément (premier en ordre inverse)
rend()	Pointe sur l'élément suivant le dernier en ordre inverse (avant le premier)

Accès aux éléments

front()	Premier élément
back()	Dernier élément
[]	Index en accès non contrôlé, non applicable aux listes
at()	Index en accès contrôlé, non applicable aux listes

Opérations sur pile et file

push_back()	Ajoute à la fin de la séquence
pop_back()	Supprime le dernier élément
push_front()	Insère un nouveau premier élément (applicable à **list** et **deque**)
pop_front()	Supprime le premier (uniquement **list** et **deque**)

Opération sur listes

insert(p)	Insère avant p
erase(p)	Supprime p
clear()	Efface tous les éléments

Opérations associatives

operator[](k)	Accède à l'élément de clé k.
find(k)	Cherche l'élément de clé k.
lower_bound(k)	Cherche le premier élément de clé k.
upper_bound(k)	Cherche le premier élément de clé supérieure à k.
equal_range(k)	Cherche tous les éléments **lower_bound** et **upper_bound** de clé k.

b. Séquences

Les séquences désignent les conteneurs qui suivent le modèle de **vector**. On trouve comme séquences les classes **vector**, **list** et **deque**.

list

Une liste est une séquence optimisée pour l'insertion et la suppression d'éléments. Elle dispose d'itérateurs bidirectionnels mais pas d'un accès par index, à la fois pour respecter la forme des algorithmes spécifiques aux listes et pour l'optimisation des performances.

La classe **list** est probablement implémentée à l'aide de listes doublement chaînées, mais le programmeur n'a pas besoin de connaître ce détail.

En supplément des opérations applicables aux séquences générales, la classe **list** propose les méthodes **splice()**, **merge()** et **sort()**.

splice()	Déplace les éléments d'une liste vers une autre, sans les copier
merge()	Fusionne deux listes
sort()	Trie une liste

D'autres méthodes sont particulièrement utiles telles **reverse()**, **remove()**, **remove_if()** et **unique()**.

Nous donnons un exemple de mise en œuvre pour certaines de ces méthodes :

```
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;

bool initiale_b(string s)
{
  return s[0]=='B';
}

int main(int argc, char* argv[])
{
  // une liste de chaînes
  list<string> liste;

  // quelques entrées
  liste.push_back("Beethoven");
  liste.push_back("Bach");
  liste.push_back("Brahms");
  liste.push_back("Haydn");
  liste.push_back("Rachmaninov");

  // retourner la liste
  liste.reverse();

  // supprimer tous les compositeurs commençant par B
  liste.remove_if(initiale_b);

  // afficher la liste
  list<string>::iterator i;
  for(i=liste.begin(); i!=liste.end() ; i++)
  {
    string nom=*i;
    cout << " " << nom.c_str();
  }
  return 0;
}
```

deque

La classe **deque** ressemble beaucoup à la classe **list**, sauf qu'elle est optimisée pour l'insertion et l'extraction aux extrémités de la séquence.

Pour les opérations d'insertion portant sur le milieu de la séquence, l'efficacité est en deçà de ce qu'offre la liste.

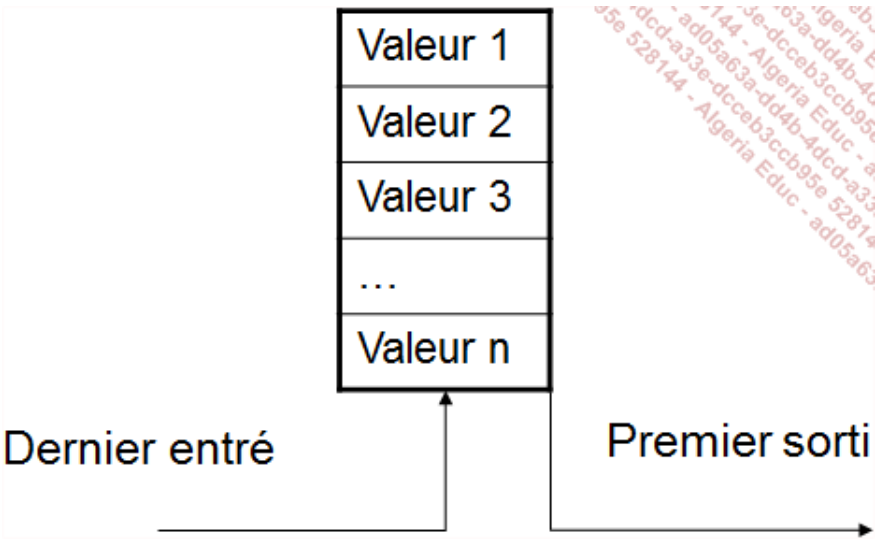
c. Adaptateurs de séquences

À partir des séquences **`vector`**, **`list`** et **`deque`** on conçoit de nouvelles séquences **`stack`**, **`queue`** et **`priority_queue`**. Toutefois, pour des raisons de performances, les classes correspondantes refondent l'implémentation. Il ne s'agit donc pas de sous-classement mais plutôt d'une adaptation. Ceci explique leur nom d'adaptateurs de séquences.

Les adaptateurs de séquence ne fournissent pas d'itérateurs, car les algorithmes qui leur sont applicables n'en ont pas besoin.

Stack

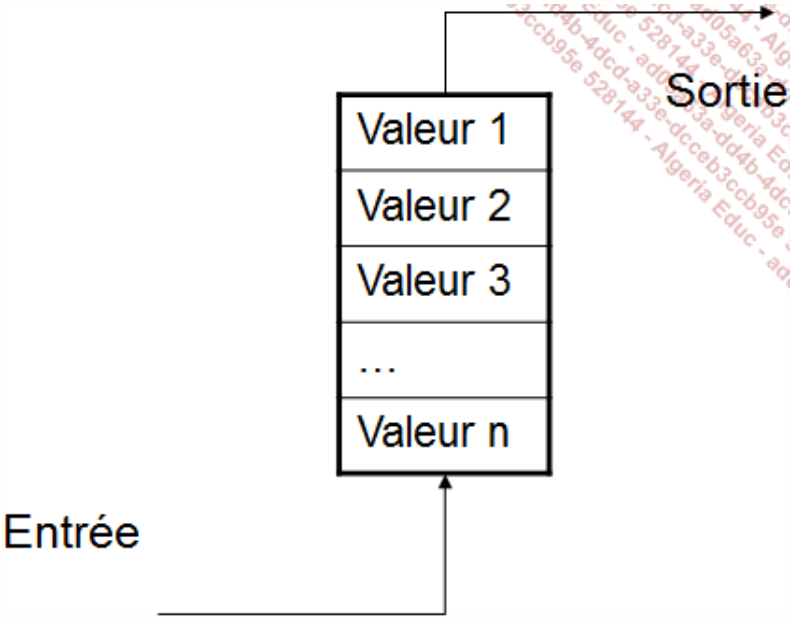
Une pile est une structure de données dynamique où il n'est pas possible d'accéder à un élément sans retirer ses successeurs. On appelle cette structure LIFO - *Last In First Out*.



L'interface **`stack`** reprend les méthodes qui ont toujours du sens pour une pile, à savoir **`back()`**, **`push_back()`** et **`pop_back()`**. Ces fonctions existent aussi sous la forme de noms plus conventionnellement utilisés en écriture d'algorithme : **`top()`**, **`push()`**, **`pop()`**.

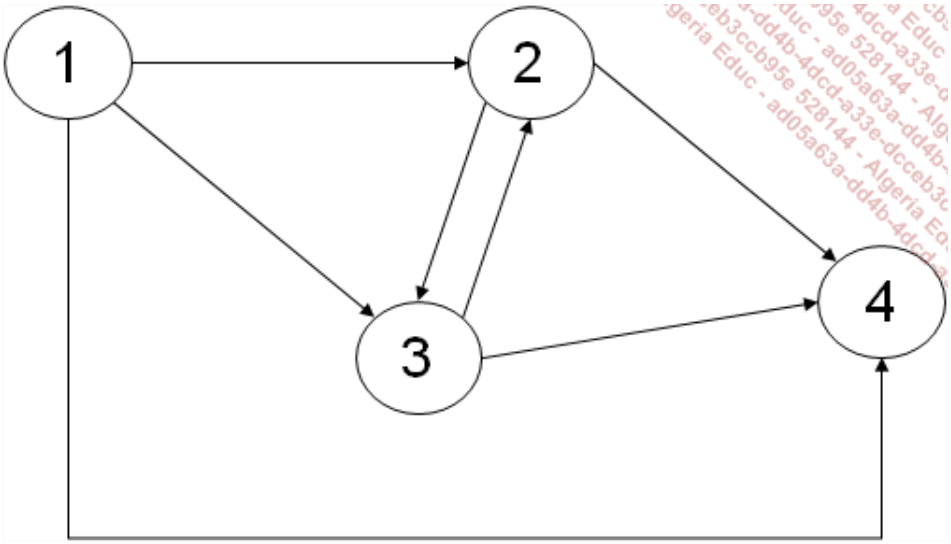
Queue

Les files d'attente sont également appelées piles FIFO - *First In First Out*. Leur fonctionnement est un peu différent des piles classiques puisqu'on insère des valeurs à la fin (empilement) et que l'on retire les valeurs au début.



La classe queue propose les méthodes d'interface **`front()`**, **`back()`**, **`push()`** et **`pop()`**. Nous proposons un petit programme de parcours de graphe écrit à l'aide d'une classe queue.

Le graphe servant d'exemple est un graphe orienté composé de 4 sommets :



Nous avons choisi d'implémenter ce graphe à l'aide d'une matrice d'adjacence, structure qui permet de conserver un algorithme très lisible :

```
#define N 4

bool**init_graphe()
{
  bool**graphe;
  graphe=new bool*[N];

  graphe[0]=new bool[N];

  graphe[0][0]=false;    // true lorsqu'il y a une adjacence
  graphe[0][1]=true;
  graphe[0][2]=true;
  graphe[0][3]=true;
```

```
  graphe[1]=new bool[N];
  graphe[1][0]=false;
  graphe[1][1]=false;
  graphe[1][2]=true;
  graphe[1][3]=true;
  graphe[2]=new bool[N];
  graphe[2][0]=false;
  graphe[2][1]=true;
  graphe[2][2]=false;
  graphe[2][2]=true;

  graphe[3]=new bool[N];
  graphe[3][0]=false;
  graphe[3][1]=false;
  graphe[3][2]=false;
  graphe[3][3]=false;

  return graphe;
}
```

Voici maintenant la structure de données qui supporte le parcours dit en largeur d'abord : une file d'attente. Comme il s'agit de parcourir un graphe pour lequel plusieurs nœuds peuvent être adjacents à d'autres nœuds, un tableau de booléens conserve l'état de visite de chaque sommet.

```
queue<int> file;
bool*sommets_visites;
```

Nous poursuivons par le code des fonctions destinées au parcours du graphe :

```
void visite(bool**graphe,int n,int sommet)
{
  if(sommets_visites[sommet])
    return;

  sommets_visites[sommet]=true;
  cout << "Visite du sommet" << (sommet+1) << endl;

  // cherche toutes les adjacences non visitées
  for(int i=0; i<n; i++)
    if(graphe[sommet][i] && ! sommets_visites[i])
      file.push(i);
}

void parcours_largeur(bool**graphe,int n)
{
  sommets_visites=new bool[n];
  for(int i=0; i<n; i++)
    sommets_visites[i]=false;  // sommet pas visité

  file.push(0);      // part du sommet 1

  while(! file.empty())
  {
    int s=file.front();
    file.pop();
    visite(graphe,n,s);
  }
}
```

Notre exemple se termine par la fonction **`main()`** et par son exécution.

```
int main(int argc, char* argv[])
{
 bool**graphe;
 graphe=init_graphe();
```

```
 parcours_largeur(graphe,N);
 return 0;
}
```

L'exécution assure bien que chaque sommet accessible depuis le sommet 1 est visité une et une seule fois :

File d'attente à priorité

Il s'agit d'une structure de données importante pour la détermination du meilleur chemin dans un graphe. L'algorithme A* (A Star) utilise justement des files d'attente à priorité. Par rapport à une file d'attente ordinaire, une pondération accélère le dépilement d'éléments au détriment d'autres.

```
priority_queue<int> pq;
  pq.push(30);
  pq.push(10);
  pq.push(20);

  cout << pq.top() << endl;  // affiche 30
```

d. Conteneurs associatifs

Les conteneurs associatifs sont très utiles à l'implémentation d'algorithmes de toutes sortes. Ces structures de données sont si pratiques que certains langages/environnements les proposent en standard. La classe principale s'appelle **map**, on la trouve parfois sous le nom de tableau associatif, ou de table de hachage.

Map

La classe **map** est une séquence de clés et de valeurs. Les clés doivent être comparables pour garantir de bonnes performances. Lorsque l'ordre des éléments est difficile à déterminer, la classe **hash_map** fournira de meilleures performances.

Les itérateurs de la classe **map** fournissent des instances de la classe pair regroupant une clé et une valeur.

```
#include <iostream>
#include <map>
using namespace std;

int main(int argc, char* argv[])
{
  map<int,string> tab;
  tab[1]="Paris";
  tab[5]="Londres";
  tab[10]="Berlin";

  map<int,string>::iterator paires;
  for(paires=tab.begin(); paires!=tab.end(); paires++)
  {
    int cle=paires->first;
    string valeur=paires->second;
    cout << cle << " " << valeur.c_str() << endl;
  }
  return 0;
```

```
}
```

Multimap

Le **multimap** est une table **map** dans laquelle la duplication de clé est autorisée.

Set

Un jeu (set) est une table **map** ne gérant que les clés. Les valeurs n'ont aucune relation entre elles.

Multiset

Il s'agit d'un set pour lequel la duplication de clé est autorisée.

3. Algorithmes

Un conteneur offre déjà un résultat intéressant mais la bibliothèque standard tire sa force d'un autre aspect : les conteneurs sont associés à des fonctions générales, des algorithmes. Ces fonctions utilisent presque toutes des itérateurs pour harmoniser l'accès aux données d'un type de conteneur à l'autre.

a. Opérations de séquence sans modification

Il s'agit principalement d'algorithmes de recherche et de parcours.

for_each()	Exécute l'action pour chaque élément d'une séquence.
find()	Recherche la première occurrence d'une valeur.
find_if()	Recherche la première correspondance d'un prédicat.
find_first_of()	Recherche dans une séquence une valeur provenant d'une autre.
adjacent_find()	Recherche une paire adjacente de valeurs.
count()	Compte les occurrences d'une valeur.
count_if()	Compte les correspondances d'un prédicat.
mismatch()	Recherche les premiers éléments pour lesquels deux séquences diffèrent.
equal()	Vaut **true** si les éléments de deux séquences sont égaux au niveau des paires.
search()	Recherche la première occurrence d'une séquence en tant que sous-séquence.
find_end()	Recherche la dernière occurrence d'une séquence en tant que sous-séquence.
search_n()	Recherche la nième occurrence d'une valeur.

Nous proposons un exemple de recherche de valeur dans un vecteur de chaînes **char*** :

```
#include <iostream>
#include <algorithm>
```

```
#include <vector>
using namespace std;

int main(int argc, char* argv[])
{
  vector<char*> tab;
  tab.push_back("Sonate");
  tab.push_back("Partita");
  tab.push_back("Cantate");
  tab.push_back("Concerto");
  tab.push_back("Symphonie");
  tab.push_back("Aria");
  tab.push_back("Prélude");

  vector<char*>::iterator pos;
  char*mot="Concerto";
  pos=find(tab.begin(),tab.end(),mot);
  cout << *pos << endl; // affiche Concerto
  return 0;
}
```

b. Opérations de séquence avec modification

transform()	Application d'une opération sur tous les éléments d'une séquence.
copy()	Copie une séquence.
copy_backward()	Copie une séquence en ordre inverse, à partir de son dernier élément.
swap()	Échange deux éléments.
iter_swap()	Idem sur la base d'itérateurs.
replace()	Remplace les éléments par une valeur donnée.
replace_if()	Remplace les éléments par une valeur donnée lorsqu'un prédicat est vérifié.
replace_copy()	Copie une séquence en remplaçant les éléments par une valeur donnée.
replace_copy_if()	Idem sur la base d'un prédicat.
fill()	Remplace chaque élément par une valeur donnée.
fill_n()	Limite l'opération fill aux n premiers éléments.
generate()	Remplace tous les éléments par le résultat d'une opération.
remove()	Supprime les éléments ayant une valeur donnée.
remove_if()	Idem sur la base d'un prédicat.
remove_copy()	Copie une séquence en supprimant les éléments d'une valeur donnée.
remove_copy_if()	Idem sur la base d'un prédicat.
unique()	Supprime les éléments adjacents et égaux.
unique_copy()	Copie une séquence en supprimant les éléments adjacents et égaux.
reverse()	Inverse l'ordre des éléments d'une séquence.
reverse_copy()	Copie une séquence en ordre inverse.

rotate()	Opère un décalage circulaire des éléments.
rotate_copy()	Idem sur la base d'une copie.
random_shuffle()	Déplace les éléments de manière aléatoire.

c. Séquences triées

sort()	Trie les éléments.
stable_sort()	Trie en conservant l'ordre des éléments égaux.
partial_sort()	Trie la première partie d'une séquence.
partial_sort_copy()	Idem sur la base d'une copie.
nth_element()	Place le nième élément à l'emplacement approprié.
lower_bound()	Recherche la première occurrence d'une valeur.
upper_bound()	Recherche la dernière occurrence d'une valeur.
equal_range()	Recherche une sous-séquence d'une valeur donnée.
binary_search()	Recherche une valeur dans une séquence triée.
merge()	Fusionne deux séquences triées.
inplace_merge()	Fusionne deux sous-séquences consécutives triées.
partition()	Déplace les éléments autour d'une valeur pivot.
stable_partition()	Identique à l'algorithme précédent, mais conserve l'ordre des éléments égaux.

d. Algorithmes de définition

includest()	Vérifie si une séquence est une sous-séquence d'une autre.
set_union()	Réalise une union triée de plusieurs séquences.
set_intersection()	Intersection triée.
set_difference()	Construit une séquence d'éléments triés dans la première, mais pas dans la deuxième.

e. Minimum et maximum

min_element()	La plus petite valeur dans une séquence.
min()	La plus petite des deux valeurs.
max_element()	La plus grande valeur dans une séquence.
max()	La plus grande des deux valeurs.

4. Calcul numérique

Le langage C++ est particulièrement attrayant pour la clarté de son exécution, dont la durée peut être prévue. Les types habituels - du **char** au **double** - sont également connus d'autres langages, comme le C ou le Pascal, et ils assurent une grande partie des calculs nécessaires à l'exécution d'un programme.

Il existe des situations pour lesquelles ces types ne sont plus adaptés, soit la précision n'est pas suffisante - même avec un **double** - soit la rapidité des calculs est mise à mal par une volumétrie trop conséquente. Les logiciels nécessitant d'importants calculs en haute précision ont du mal à satisfaire une optimisation avancée des performances avec une représentation satisfaisante des valeurs numériques.

Dressons un rapide état des lieux des possibilités de calcul numérique en C++ avant d'introduire la partie correspondante dans la bibliothèque standard.

a. Limites des formats ordinaires

L'en-tête **<limits>** offre un certain nombre de classes paramétrées pour déterminer les valeurs caractéristiques d'un format tel short ou double. Quelle est la plus petite valeur, quel est le plus petit incrément (epsilon)...

Le programme suivant nous donne des informations sur le type double :

```
#include <iostream>
#include <limits>
using namespace std;
int main(int argc, char* argv[])
{
  cout << "Plus petite valeur (double)=" <<
numeric_limits<double>::min() << endl;

  cout << "Plus grande valeur (double)=" <<
numeric_limits<double>::max() << endl;
  cout << "Plus petite valeur absolue significative (double)=" <<
numeric_limits<double>::epsilon() << endl;
  cout << "Représentation d'une quantité qui n'est pas un nombre
(double)=" << numeric_limits<double>::signaling_NaN() << endl;
  return 0;
}
```

b. Fonctions de la bibliothèque

Les en-têtes de la bibliothèque du C, **<cmath>** et **<math.h>** fournissent le support pour les fonctions mathématiques usuelles, applicables pour la plupart au type double. Ce type a été choisi d'une part parce qu'il correspond à un format standardisé (IEEE 754), et d'autre part car les microprocesseurs savent travailler directement avec ce format, sans prendre plus de temps que le **float** qui offre lui une moins bonne précision. Attention toutefois à certaines implémentations logicielles qui peuvent faire varier légèrement ces formats.

Il peut être intéressant de consulter l'excellent ouvrage de Laurent Padjasek, Calcul numérique en assembleur (Sybex) pour découvrir toutes les subtilités du codage IEEE 754.

Nous proposons maintenant une liste des principales fonctions de la bibliothèque mathématique du C.

abs(),fabs()	valeur absolue
ceil()	plus petit entier non inférieur à l'argument
floor()	plus grand entier non supérieur à l'argument
sqrt()	racine carrée

pow()	puissance
cos(), sin(), tan()	fonctions trigonométriques
acos(), asin(), atan()	arc cosinus, arc sinus, arc tangente, fonctions trigonométriques inverses
atan2(x,y)	atan(x/y)
sinh(), cosh(), tanh()	fonctions trigonométriques hyperboliques
exp(), log(), log10()	exponentielle et logarithmes
mod()	partie fractionnaire

c. Initiatives complémentaires

Puisque C++ a été conçu pour pouvoir créer des nouveaux types de données, certains travaux non officiellement reconnus par la librairie standard fournissent de très bons résultats dans beaucoup de domaines comme l'ingénierie, le calcul financier où le calcul numérique. Citons pour l'exemple les travaux d'Alain Reverchon et de Marc Ducamp, synthétisés dans l'ouvrage Outils mathématiques pour C++ (Armand Colin).

d. Fonctions de la bibliothèque standard et classe valarray

La bibliothèque standard s'est surtout attachée à proposer un mécanisme qui assure la jointure entre l'optimisation des calculs et le programme proprement dit. Les microprocesseurs adoptent le travail à la chaîne et en parallèle pour optimiser les calculs, concept que l'on désigne sous le terme de calcul vectoriel.

Nous trouvons dans la S.T.L. la structure **valarray** dont le rôle est justement de préparer l'optimisation des calculs tout en conservant une assez bonne lisibilité du programme.

Cette classe possède parmi ses constructeurs une signature qui accepte un tableau ordinaire, initialisation qui assure une bonne intégration avec l'existant.

Le programme suivant crée un **valarray** à partir d'une série de valeurs doubles fournies dans un tableau initialisé en extension. Une seule opération **v+=10** correspond à une addition effectuée sur chaque valeur du vecteur. Suivant les implémentations et les capacités du microprocesseur, cette écriture raccourcie ira de pair avec une exécution très rapide. Il faut d'ailleurs remarquer que les processeurs RISC tirent pleinement parti de ce type d'opération.

```
#include <iostream>
#include <valarray>
using namespace std;

int main(int argc, char* argv[])
{
  double d[]={3, 5.2, 8, 4 };
  valarray<double> v(d,4);
  v+=10;    // ajoute 10 à chaque élément

  return 0;
}
```

La bibliothèque standard offre également le support des matrices (**slice_array**) formées à partir de **valarray** et après une étape de structuration appelée slice.

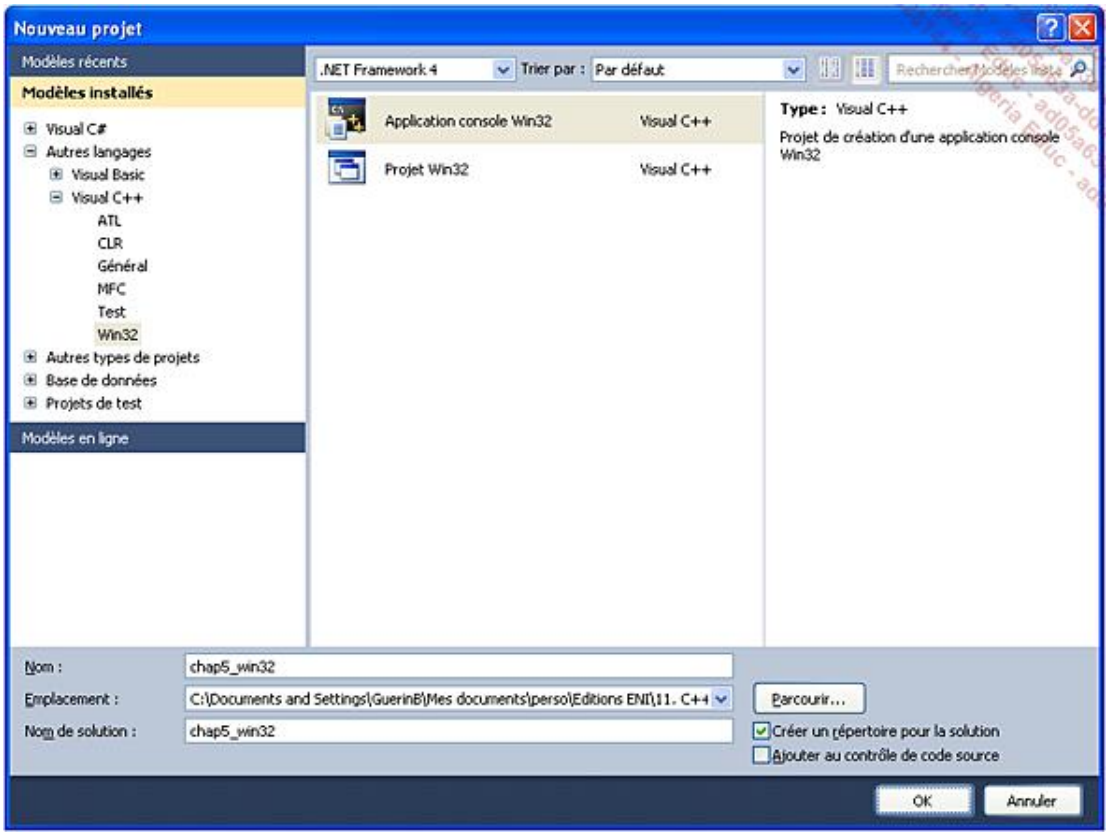
L’environnement Windows

1. Les programmes Win32

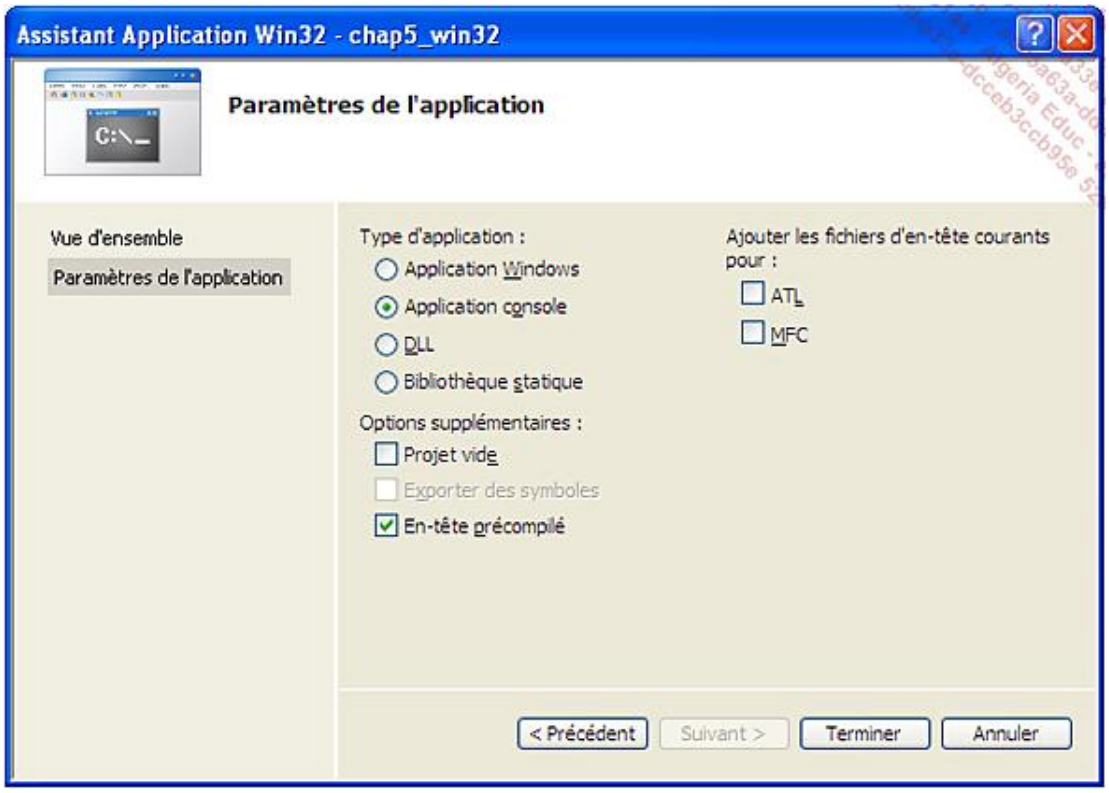
La plate-forme Win32 a très vite été supportée par C++ car Microsoft a lancé dès 1992 l’un des premiers environnements intégrés pour ce langage ; il s’agissait de Visual C++ et à cette époque, de nombreux architectes étaient convaincus de l’intérêt d’un langage objet proche du langage C pour créer des applications fonctionnant dans des environnements graphiques.

Toutefois, la programmation d’une application fenêtrée, sans framework, n’est pas une tâche aisée. Il n’en demeure pas moins que la plateforme Win32 propose plusieurs formats d’applications : applications "console", services Windows, bibliothèques dynamiques (DLL) et bien entendu les applications graphiques fenêtrées.

Visual C++ 2010 dispose d’un assistant pour démarrer un projet de type Win32 :



L’assistant donne le choix pour le format définitif du projet : application console ou graphique (Windows), DLL, bibliothèque statique… On peut également raccorder le projet aux frameworks ATL et MFC, mais nous verrons ce dernier un peu plus loin.



Les en-têtes précompilés réduisent le temps de compilation, car les volumineux fichiers d'en-têtes standards comme **windows.h** (plus de 100000 lignes), sont analysés avant de démarrer la phase de compilation. En conséquence pour le programmeur, tous les fichiers de code **.cpp** doivent rappeler le fichier d'en-tête **stdafx.h** :

```
// chap5_win32.cpp : définit le point d'entrée pour l'application console.
//

#include "stdafx.h"


int _tmain(int argc, _TCHAR* argv[])
{
    return 0;
}
```

La macro **_TCHAR** désigne en fait le type **wchar_t** qui est assimilable à un caractère unicode.

Les applications Win32 de Visual C++ bénéficient des éléments de la norme C++ ANSI - syntaxe, STL, et autres bibliothèques standard - mais aussi des bibliothèques spécifiques à Windows.

```
// chap5_win32.cpp : définit le point d'entrée pour l'application console.
//

#include "stdafx.h"
#include <string> // STL
#include <Windows.h> // Windows

int _tmain(int argc, _TCHAR* argv[])
{
    return 0;
}
```

2. Le framework MFC

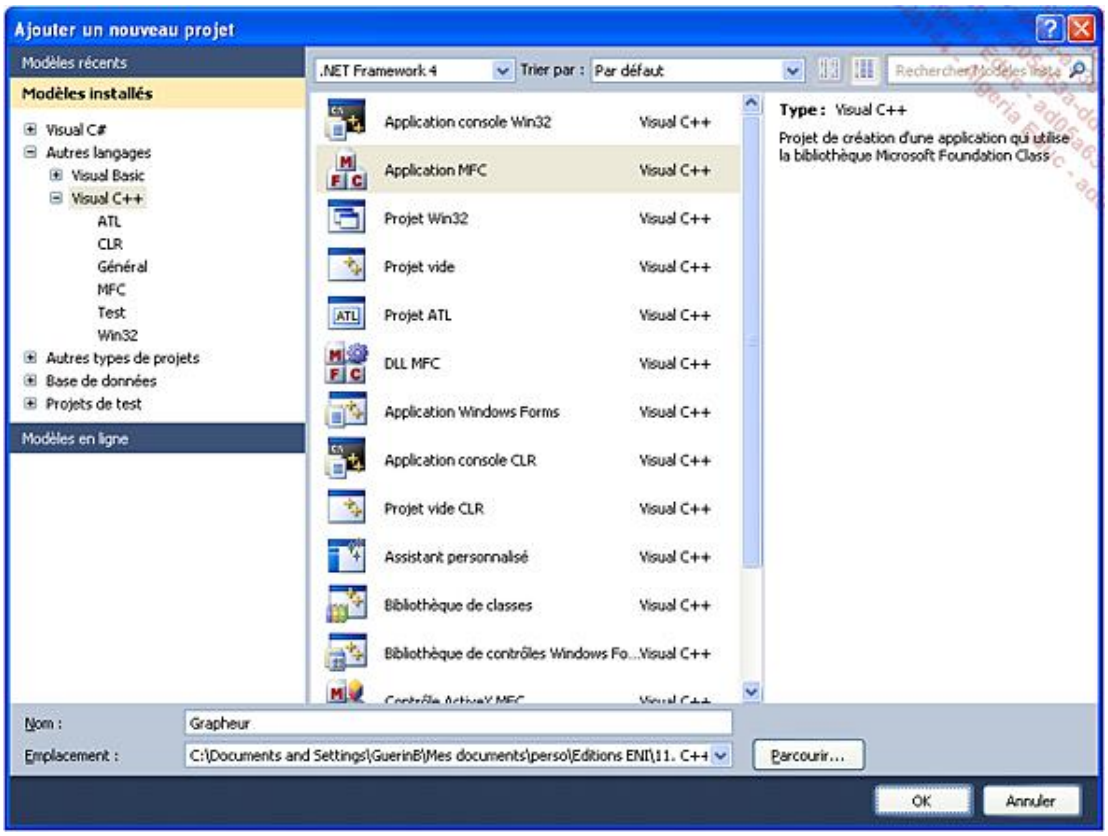
Très rapidement la programmation d'une application graphique fenêtrée en mode C s'est avérée contre-productive. Microsoft a proposé pour sa part un framework de développement, désigné à partir des trois lettres qui explicitent la nouvelle bibliothèque de classes, Microsoft Foundation Classes. L'environnement MFC est cependant bien davantage qu'une API, c'est un cadre d'application structurant et couvrant tous les aspects du développement d'applications fenêtrées :

Organisation générale du programme et séparation des différents d'application	Utilisation du design pattern MVC grâce à l'architecture Document-Vue de MFC.
Création de types de documents structurés	Sérialisation et mécanismes de définition de formats de documents
Présentations standardisées des applications	Templates d'applications SDI (simple document), MDI (multiples documents), Explorateur…
Prise en charge avancée du graphisme	Modèle objet pour tirer le meilleur parti de GDI (Graphic Device Interface)
Support des bases de données, du réseau, de formulaires HTML…	Bibliothèques de classes spécifiques et composants Active X.

Les MFC se sont étoffées au fil des années et restent très populaires auprès des développeurs C++ même si de nouveaux environnements ont fait leur apparition.

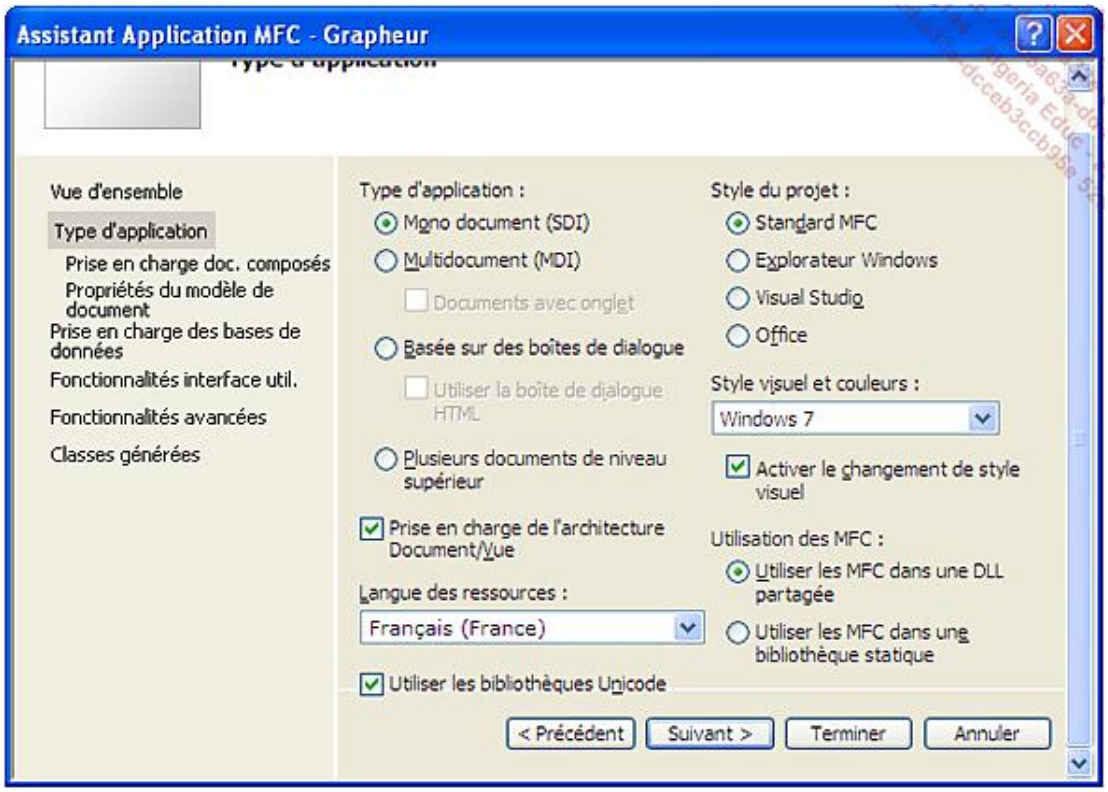
a. Création d'une application MFC

Accessible depuis le menu **Fichier - Nouveau projet**, Visual C++ dispose d'un assistant pour créer des applications MFC :

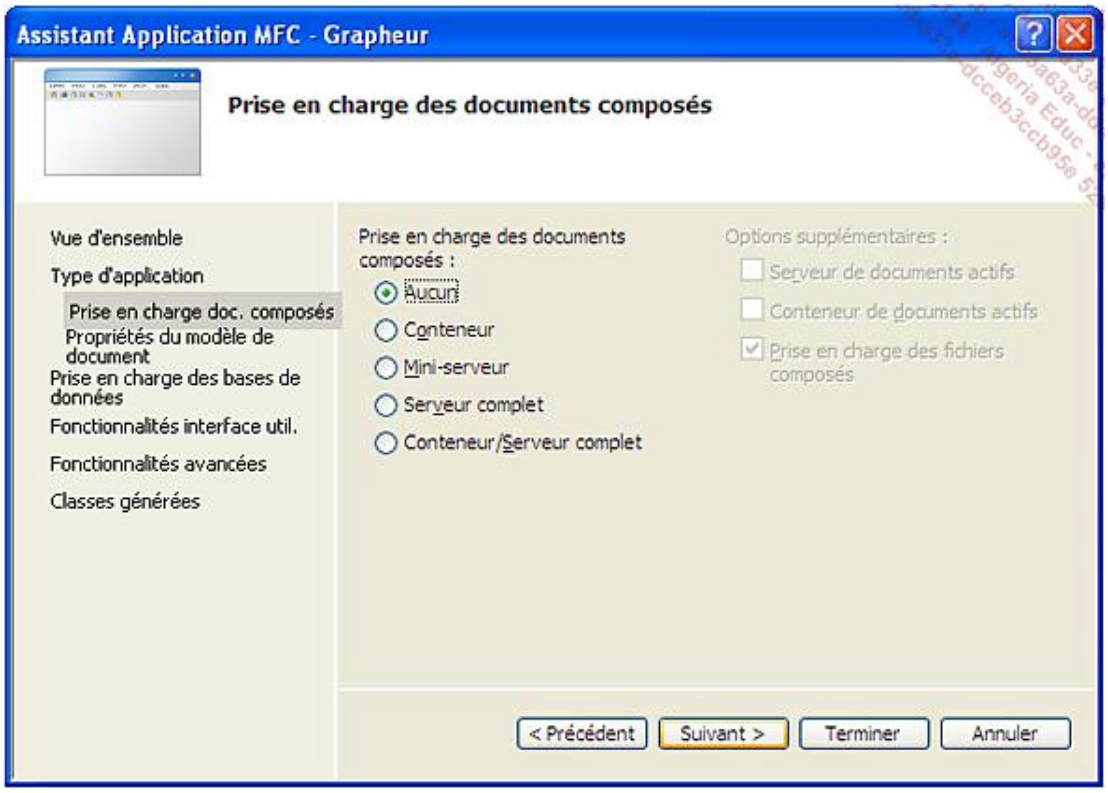


La première étape de l'assistant propose plusieurs formats d'applications : SDI - comme le bloc-notes, MDI - comme les anciennes versions de Word avec plusieurs documents ouverts et un menu fenêtre… Le style du

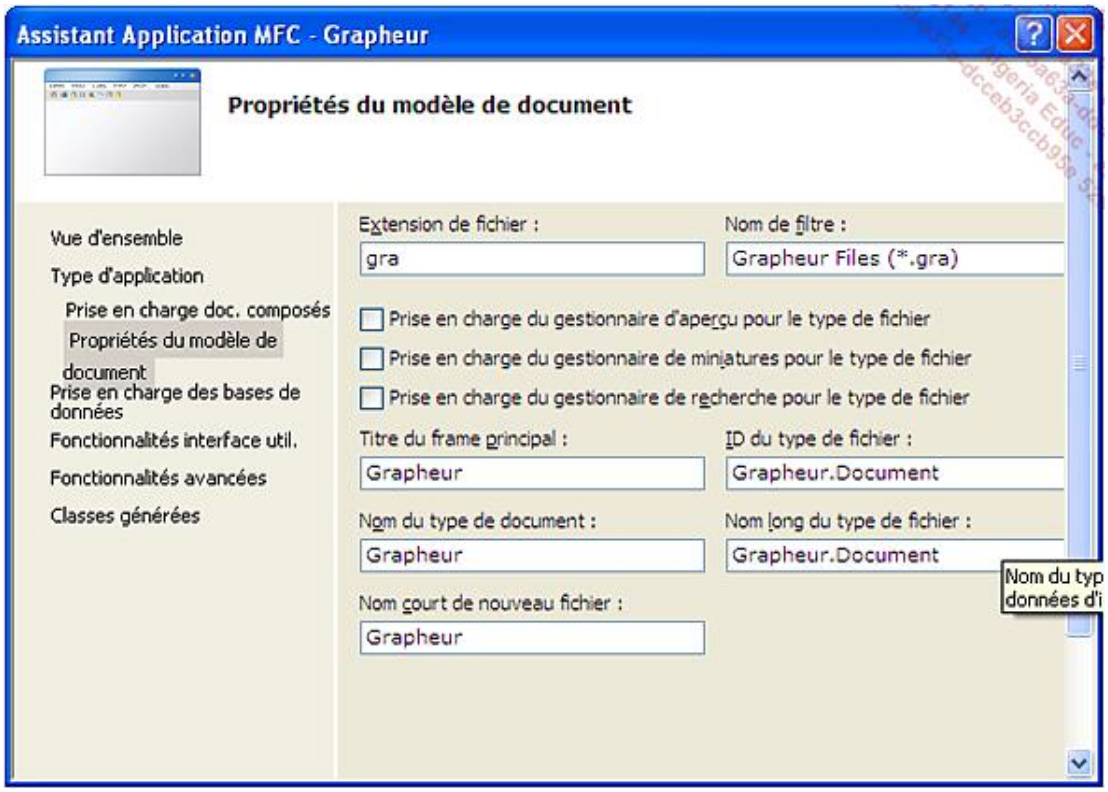
projet correspond à la présentation par défaut. Enfin les MFC peuvent être employées sous forme d'une DLL (l'exécutable est plus compact mais il faut penser à déployer les bons fichiers) ou comme une bibliothèque statique.



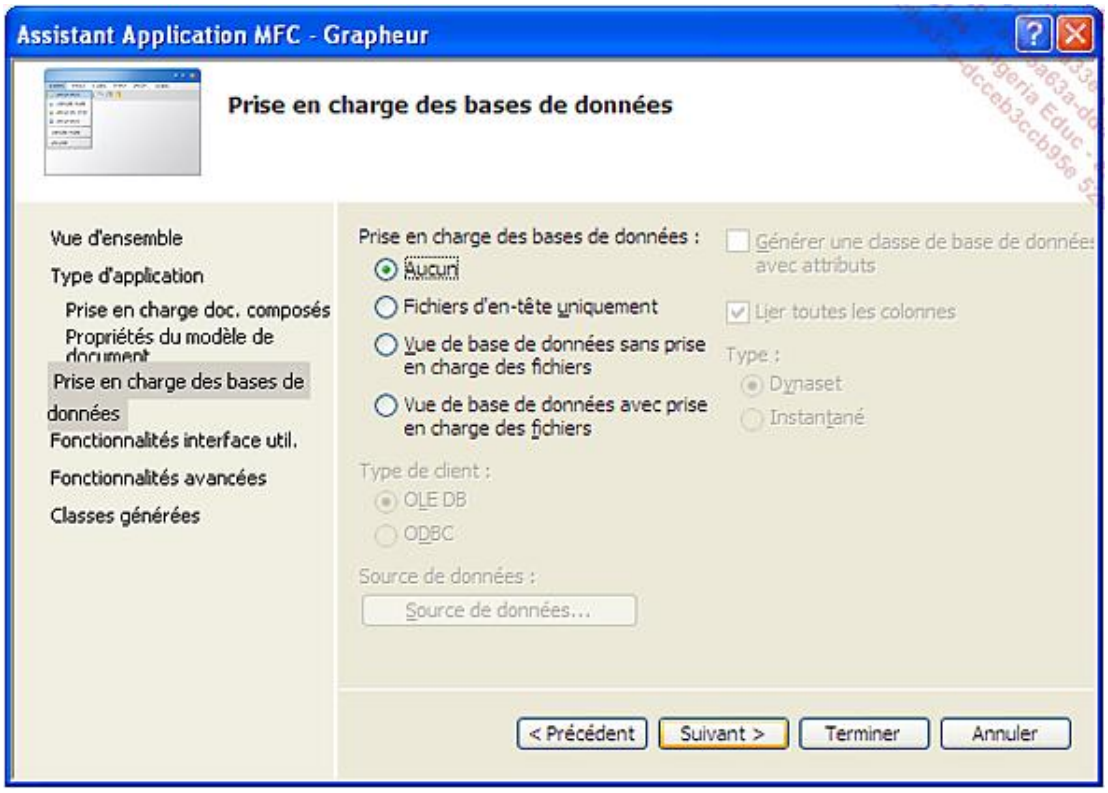
Vient ensuite le choix de supporter les documents composés - par exemple intégrer des objets dans un document Office :



Visual C++ et MFC créent une extension propre aux documents de l'application. De cette façon un double clic sur un fichier depuis l'explorateur Windows lance l'application appropriée car l'extension a été enregistrée par le système.

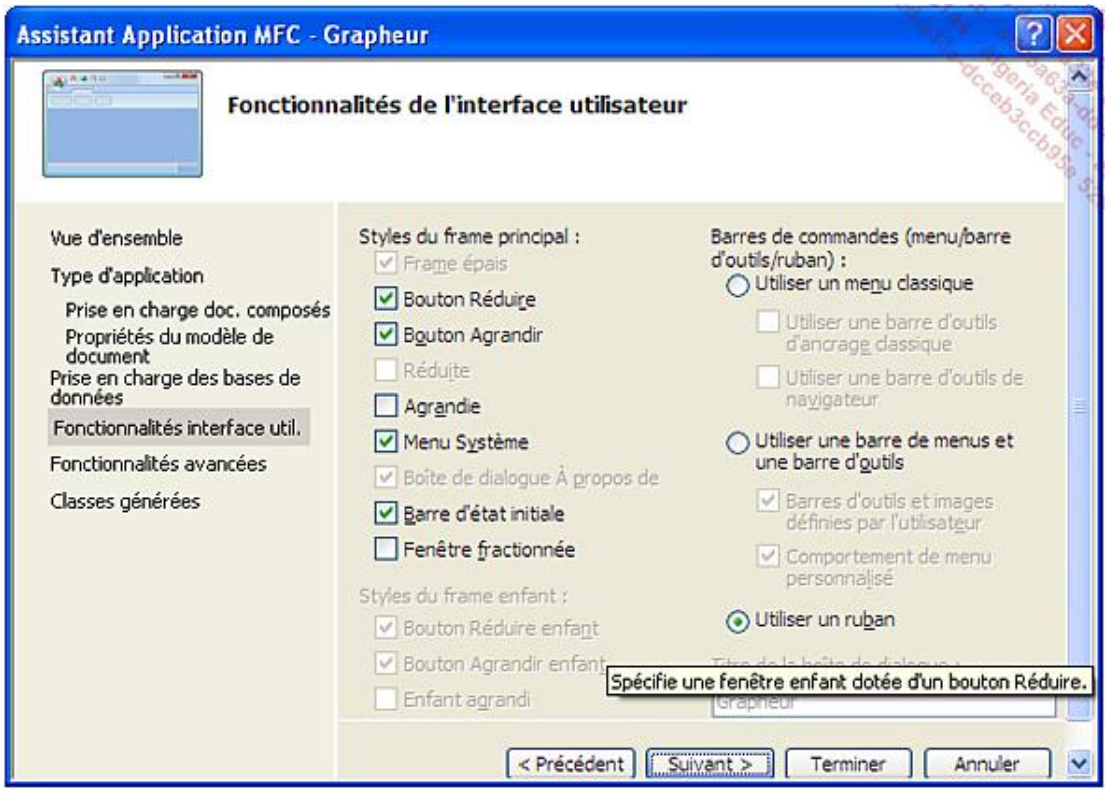


Les applications MFC bénéficient aussi du support des bases de données et plusieurs niveaux sont proposés :

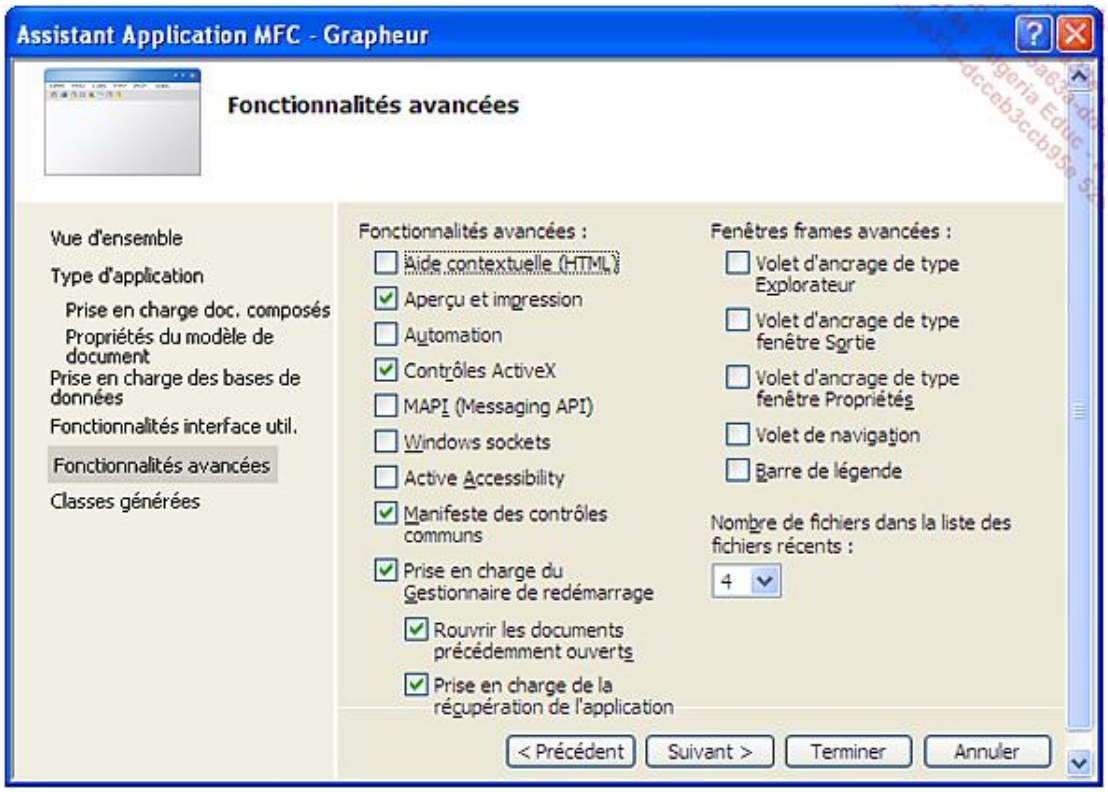


À l'étape suivante, on décide du style de la fenêtre principale et du type de menu que va recevoir le menu. Il

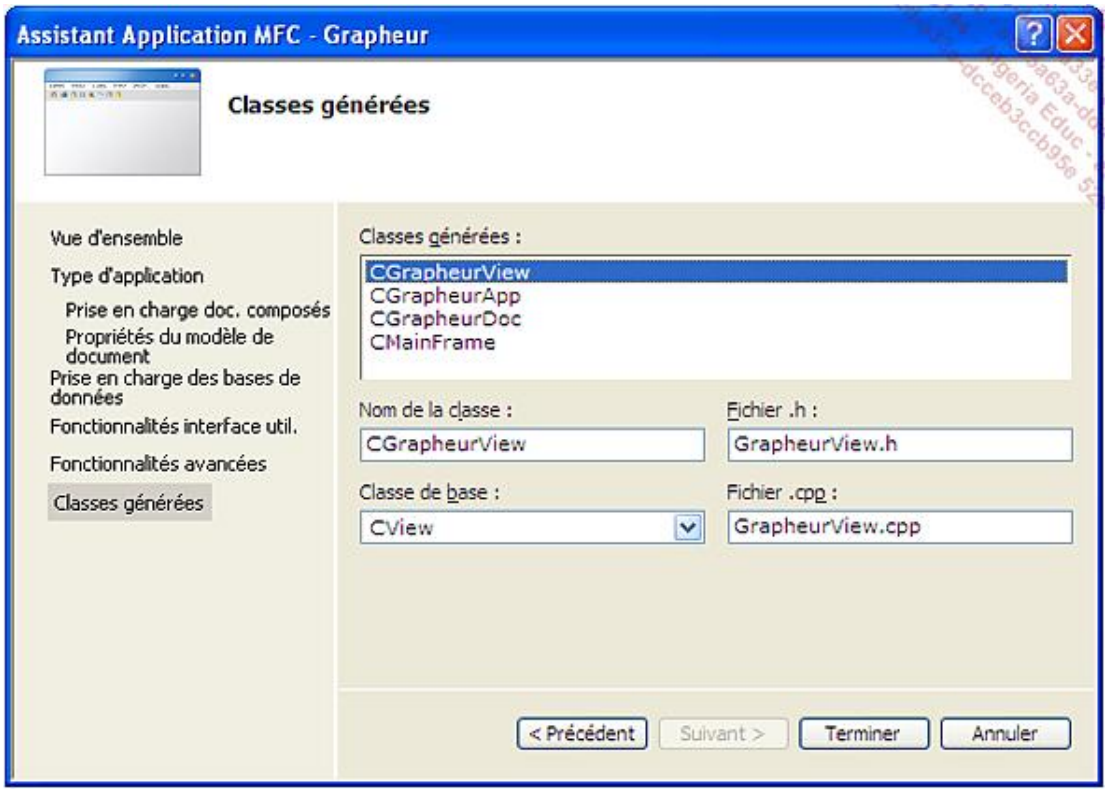
est à noter que les MFC proposent un menu de type ruban Office même hors de Windows 7.



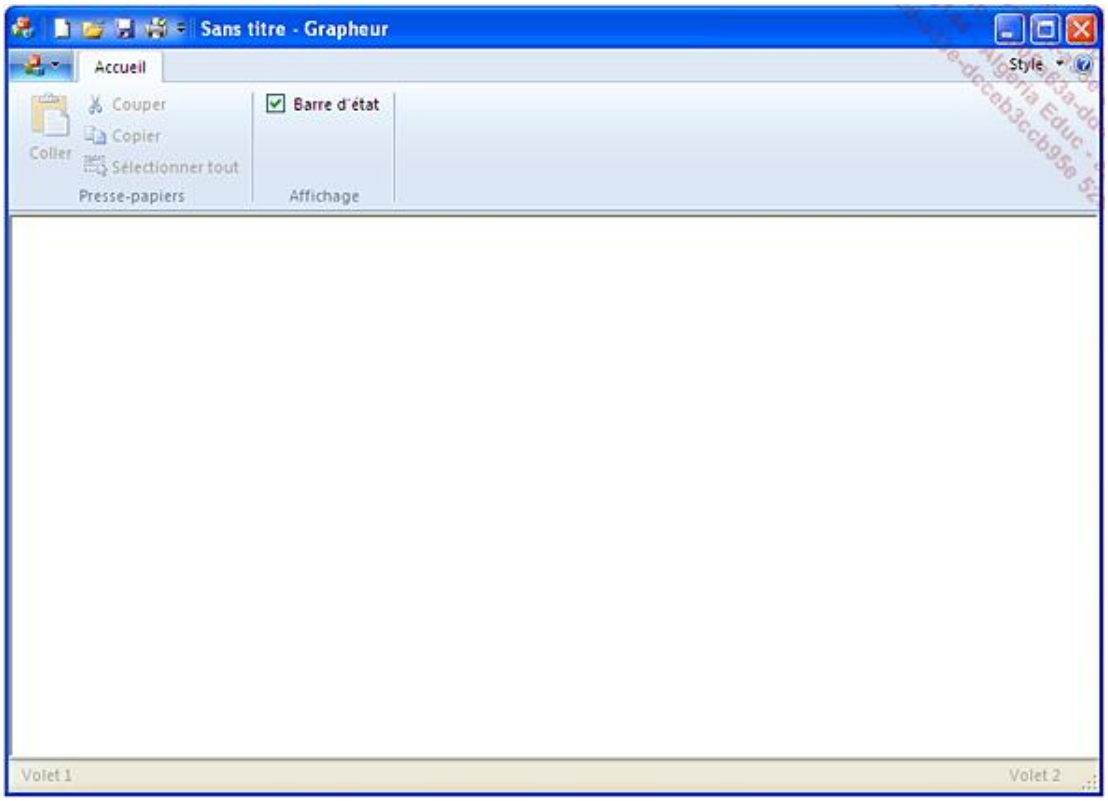
L'avant-dernière étape est également consacrée à des fonctionnalités optionnelles et avancées.



La dernière étape de l'assistant est très importante pour définir les classes composant l'application : le programmeur doit choisir leur nom et leur modèle de base - par exemple **`CView`** ou **`CScrollView`**.



Après l'assistant, une application basique correspondant aux paramètres choisis est générée par Visual Studio. On peut déjà l'exécuter pour contrôler le résultat :

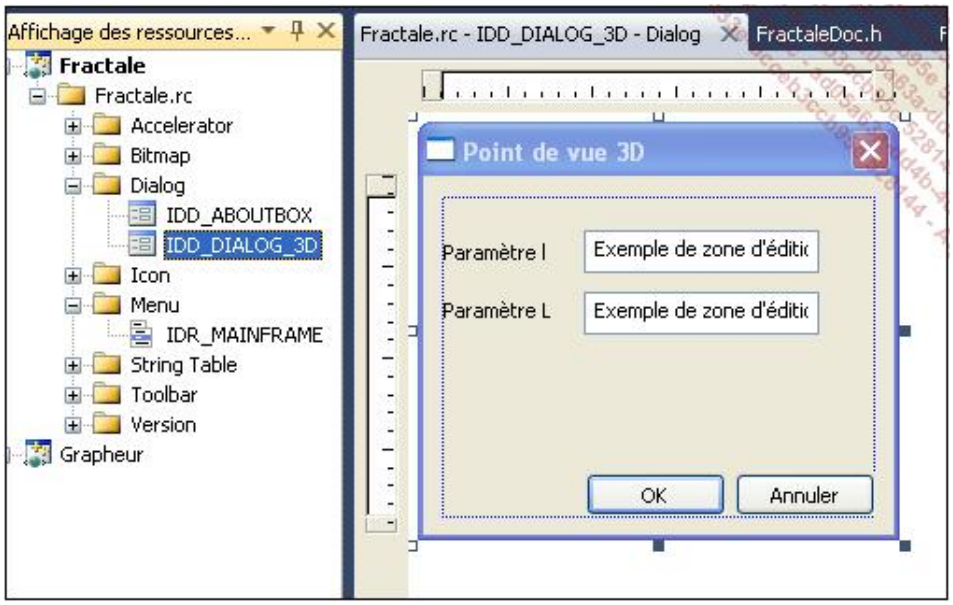


b. L'architecture Document-Vue

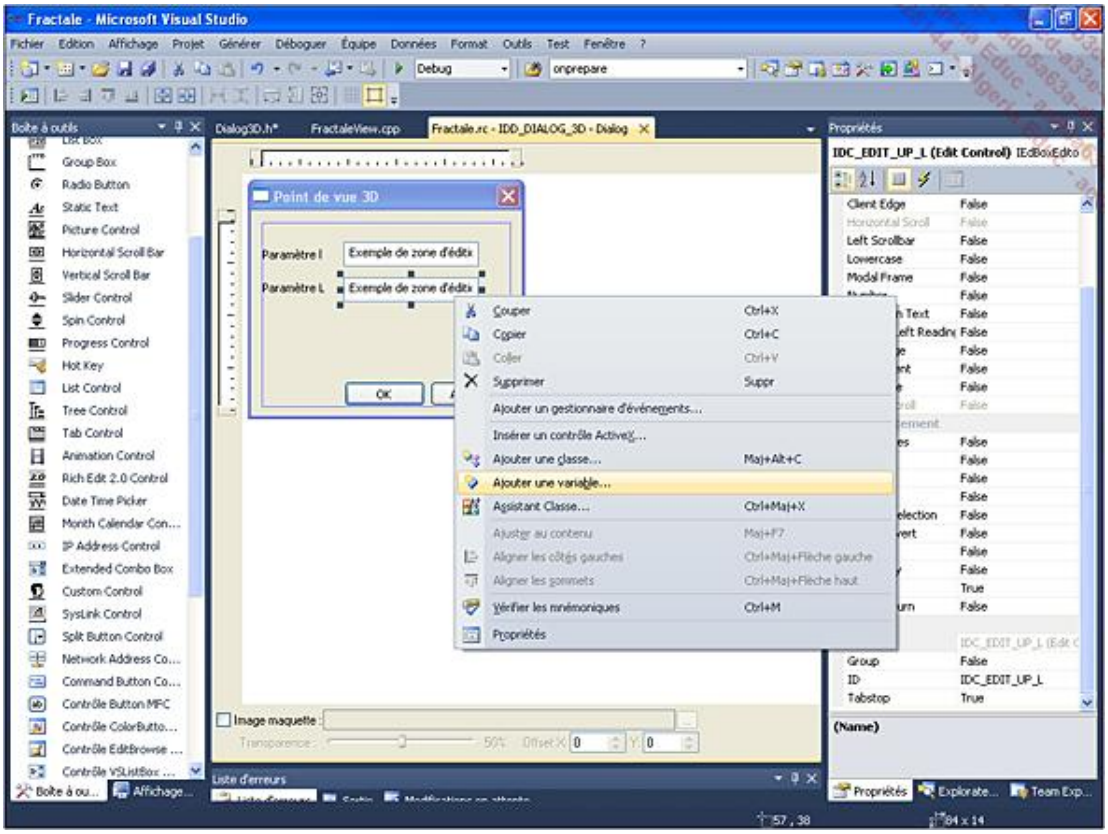
L'architecture Document-Vue guide le programmeur pour structurer son application, en séparant clairement les responsabilités de chaque classe. Pour donner un aperçu de ce design pattern, nous allons prendre l'exemple

de l'implémentation d'une boîte de dialogue servant à modifier les paramètres de projection 3D d'une figure à dessiner à l'écran.

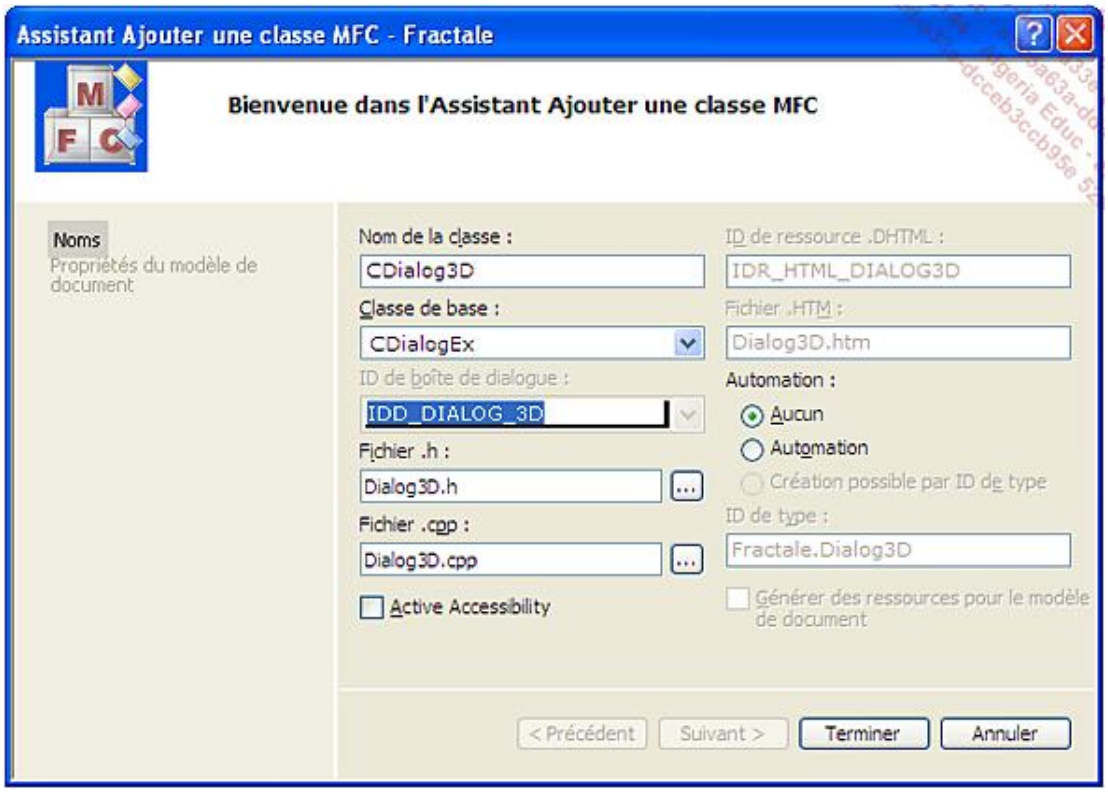
La première étape consiste à construire la boîte de dialogue depuis l'éditeur de ressources de Visual C++. On procède depuis l'explorateur de ressources et la boîte à outils.



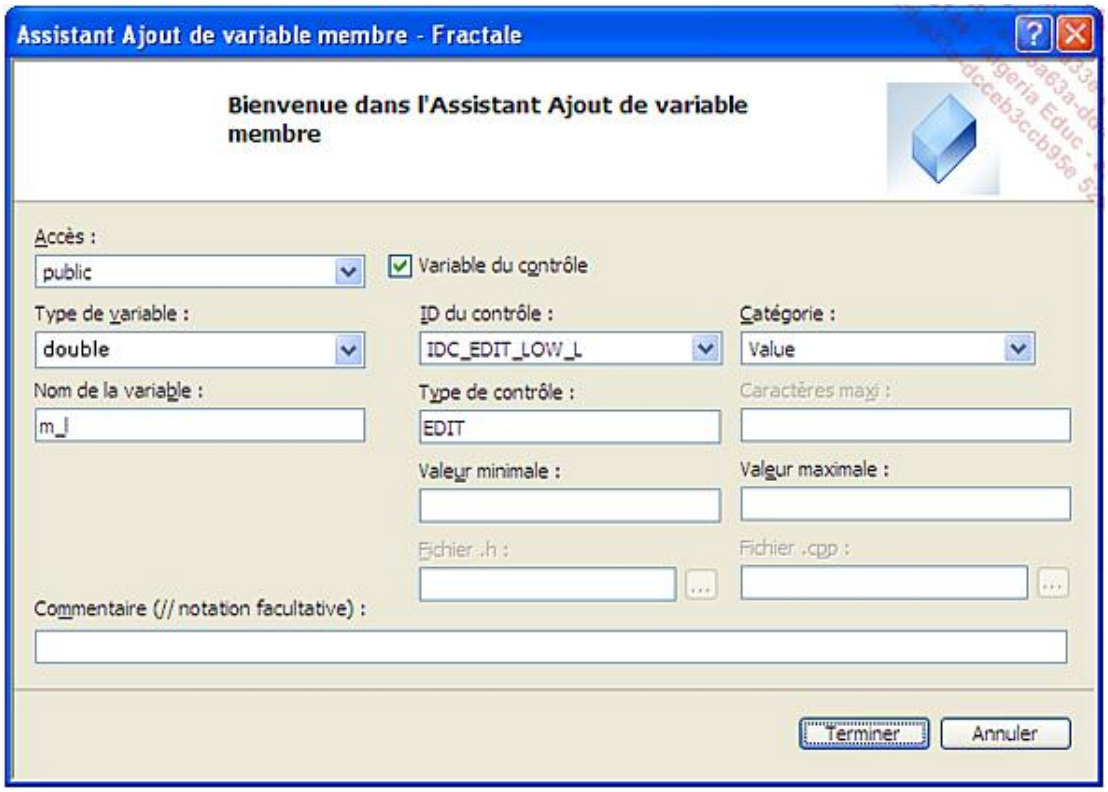
En deuxième étape, les menus contextuels **Ajouter une classe** puis **Ajouter une variable** servent respectivement à associer une classe héritant de **`CDialog`** puis à associer à chaque contrôle une variable typée :



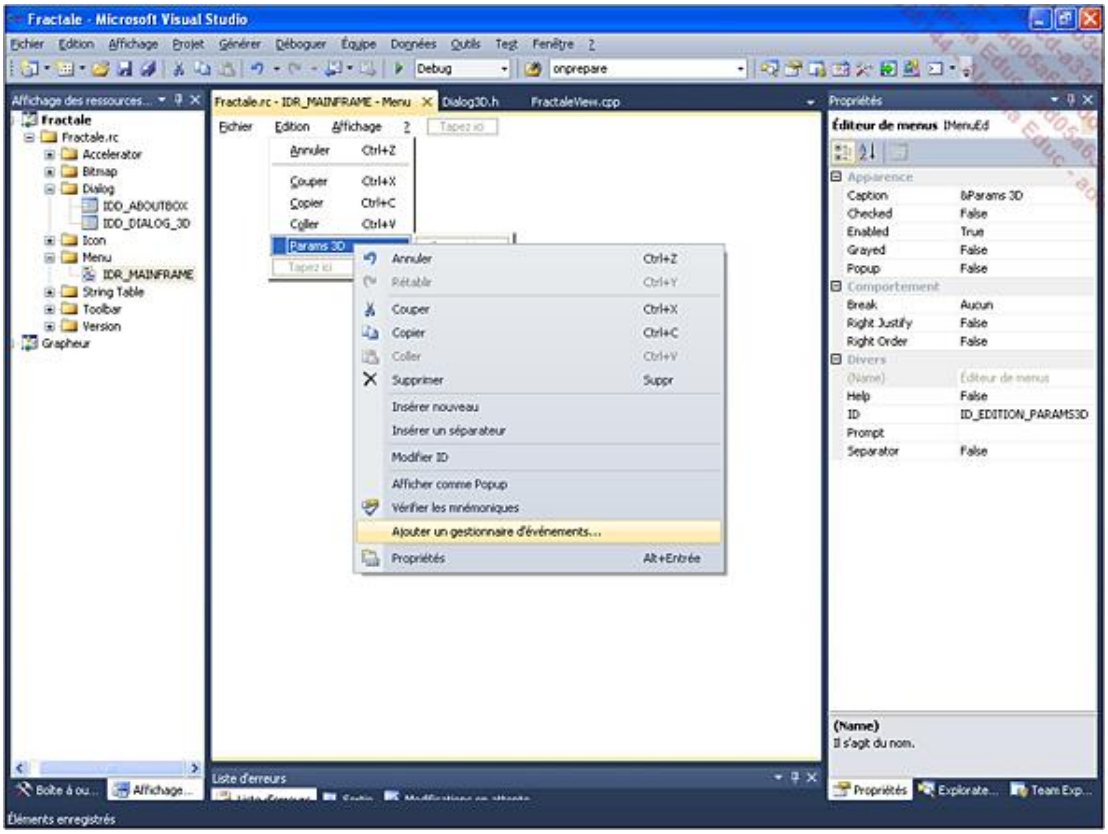
Voilà d'abord pour la définition de la classe :



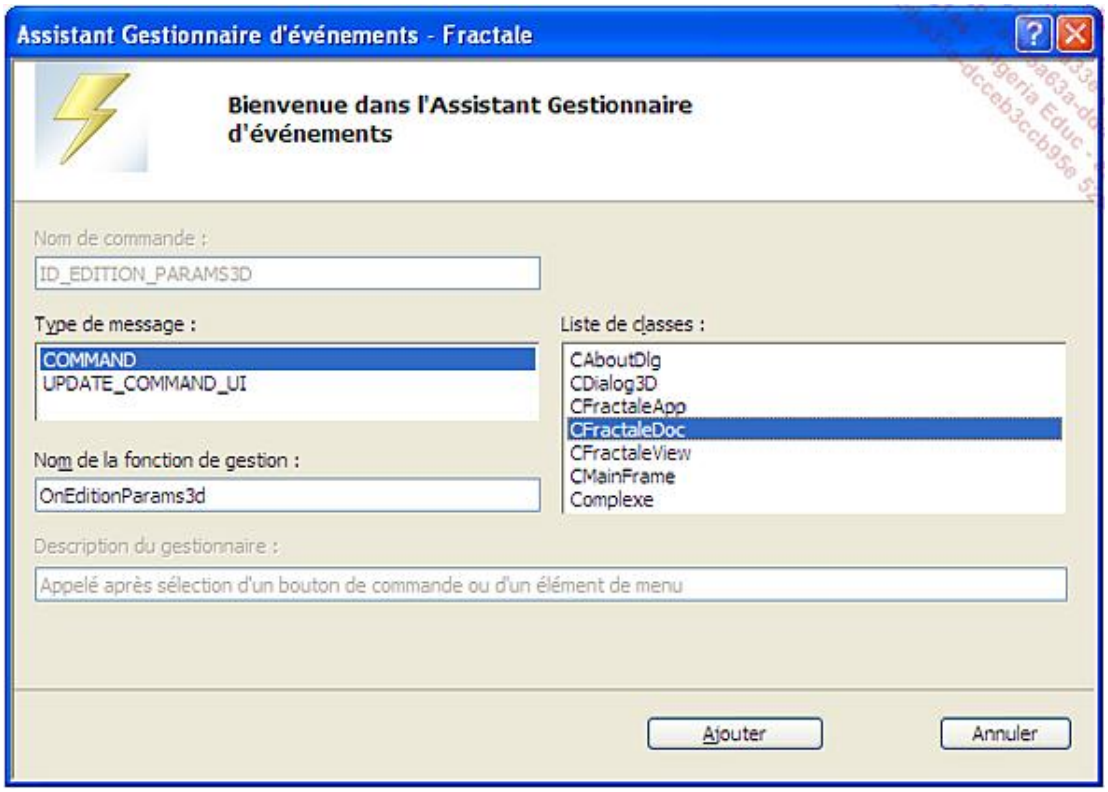
Vient ensuite l'ajout des variables. On remarquera le préfixe caractéristique des applications MFC sur le nom de la variable, **`m_`** :



La boîte de dialogue sera affichée à partir d'un menu spécifique :



Par proximité, nous décidons d'implémenter le gestionnaire d'événements commandant l'affichage de la boîte de dialogue dans la classe dérivant de **`CDocument`** :



Avant de définir ce gestionnaire d'événements, nous ajoutons dans la classe document deux variables contrôlant la perspective de la figure à dessiner. Elles seront évidemment éditées à travers la boîte de dialogue :

```
class CFractaleDoc : public Cdocument
{
// Attributes
public:
    double m_l, m_L;
```

Comme ces variables sont utilisées dès le démarrage de l'application, leur initialisation a lieu dans la procédure **OnNewDocument** appelée par le framework :

```
BOOL CfractaleDoc::OnNewDocument()
{
    if (!Cdocument::OnNewDocument())
        return FALSE;

    // valeurs "initiales" des paramètres de projection
    // s'agissant d'une application SDI, la commande Fichier / Nouveau
    // rappelle cette procédure
    m_l = 1.1;
    m_L = 1.2;

    return TRUE;
}
```

Nous poursuivons par l'implémentation du gestionnaire d'événement destiné à afficher la boîte de dialogue :

```
void CfractaleDoc::OnEditionParams3d()
{
    // instancie la boîte de dialogue
    CDialog3D dlg;

    // lecture des valeurs à modifier
    dlg.m_l = m_l;
    dlg.m_L = m_L;

    // affichage modal
    dlg.DoModal();

    // mise à jour des valeurs dans la classe Document
    m_l = dlg.m_l;
    m_L = dlg.m_L;

    // actualisation du tracé de la figure
    InvalidateRect(NULL,NULL,true);
}
```

Pour terminer, voici comment les paramètres sont lus dans la classe **Vue** dérivant de **CView**. La procédure **OnDraw** est appelée par le framework lorsqu'il faut actualiser la zone cliente de la fenêtre, autant en affichage qu'en impression :

```
void CFractaleView::OnDraw(CDC* pDC)
{
...
    l = ((CfractaleDoc*)GetDocument())->m_l;
    L = ((CfractaleDoc*)GetDocument())->m_L;
...
}
```

c. Programmation graphique

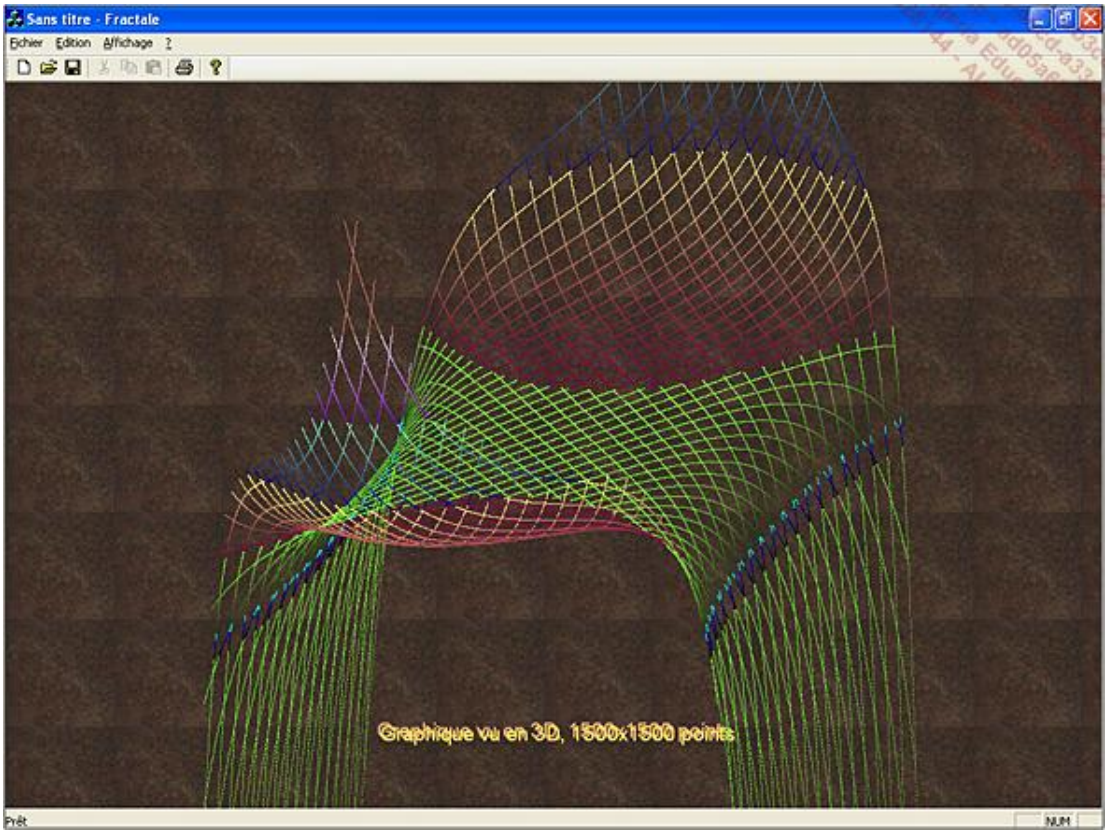
Windows et les MFC apportent au programmeur tous les outils nécessaires pour définir des applications graphiques de haut niveau. Le code suivant donne un aperçu de la mise en œuvre d'un affichage élaboré. On se rendra compte à quel point les applications graphiques peuvent s'avérer délicates à programmer :

```
void CFractaleView::OnDraw(CDC* pDC)
{

    // associer la palette au DC
    CPalette* pOldPal;        // ancienne
    pOldPal = pDC->SelectPalette(&m_Palette, true);
    pDC->RealizePalette() ;

    // MX et MY déterminent la qualité de l'image
    int MX,MY;
    MX = 1500;
    MY = 1500;

    // dimensionner l'écran
    RECT rc;
    GetClientRect(&rc);

    // changer de repère et de système de coordonnées
    if(!pDC->IsPrinting()) {
        // dessiner une image de fond
        CBrush brosse;
        CBitmap image;
        image.LoadBitmap(IDB_IMAGE);
        brosse.CreatePatternBrush(&image);
        pDC->FillRect(&rc,&brosse);

        pDC->SetMapMode(MM_ISOTROPIC);
        pDC->SetViewportExt(rc.right,-rc.bottom);
    // rapport de coordonnées
        pDC->SetViewportOrg(rc.right/2,rc.bottom/2);
    // fixer l'origine
    }
    pDC->SetWindowExt(2*MX,2*MY);
    // plan de 1000² points

    CRect zone_invalide;    // zone à rafraîchir
    CPoint test;        // un point
    pDC->GetClipBox(&zone_invalide);
    zone_invalide.NormalizeRect();

    // créer une police
    police=new Cfont;
    police->CreatePointFont(14*10,"Arial",pDC);
    pDC->SelectObject(police);

    CRect pos;
    CSize ttexte;
    char msg[500];
    sprintf(msg,"Graphique vu en 3D, %dx%d points",MX,MY);
    ttexte=pDC->GetTextExtent(msg);

    pos.left=-(ttexte.cx)/2;
    pos.right=pos.left+ttexte.cx;

    pos.top=-MX*0.75;
    pos.bottom=-MX;

    if(!pDC->IsPrinting())
    {
        pDC->SetBkMode(TRANSPARENT);
        pDC->SetTextColor(RGB(248,172,122));
        pDC->DrawText(msg,&pos,0);

        int dec = 15;
        pos.left += dec;
        pos.top -= dec;
        pos.right += dec;
```

```
        pDC->SetTextColor(RGB(250,234,96));
        pDC->DrawText(msg,&pos,0);
    } else {
        pDC->SetTextColor(RGB(0,0,0));
        pDC->DrawText(msg,&pos,0);
    }



    // ------------------------------------------------------
    // vue en 3D
    double a1,a2,a3,b1,b2,b3;
     // coefficients de projection 3D
    double l,L;                    // latitude et longitude

    // point de vue
    l = 1.1;
    L = 1.20;

    l = ((CfractaleDoc*)GetDocument())->m_l;
    L = ((CfractaleDoc*)GetDocument())->m_L;

    // ces coefficients sont calculés pour une projection orthogonale
    a1 = -sin(L);
    a2 = cos(L);
    a3 = 0;
    b1 = -sin(l)*cos(L);
    b2 = -sin(l)*cos(L);
    b3 = cos(l);

    // ------------------------------------------------------
    // tracé d'une figure mathématique en 3D
    int i,j;    // coordonnées dans l'espace logique
    int n;      // nombre d'itération
    int X,Y;    // coordonnées projetées (X,Y)=proj(x,y,z)

    double cote,x,y; // coordonnées mathématiques
    int coul;        // couleur des points

    for(i=-MX;i<MX;i+=80)
        for(j=-MY;j<MY;j++)
        {

            x = (double)i/MX*2;
            y = (double)j/MY*2;
            cote = 100 * sin(x)*sin(y)*exp(x*x - x*y +y*y);

            coul = 1000 + cote;

            if(coul>4000)
               coul = 1500;

            if(coul<0)
               coul = 800;

            // projection sur un plan
            X= a1*i + a2*j + a3*cote;
            Y= b1*i + b2*j + b3*cote;

            // test d'impact
            test.x=X;
            test.y=Y;

            if(PtInRect(&zone_invalide,test))
// le point est-il invalide ?
                pDC->SetPixel(test,PALETTEINDEX(coul));
// 40*100=4000
```

```
        }

    for(i=-MX;i<MX;i++)
        for(j=-MY;j<MY;j+=80)
        {

            x = (double)i/MX*2;
            y = (double)j/MY*2;
            cote = 100 * sin(x)*sin(y)*exp(x*x - x*y +y*y);

            coul = 1000 + cote;

            if(coul>4000)
                coul = 1500;

            if(coul<0)
                coul = 800;

            // projection sur un plan
            X= a1*i + a2*j + a3*cote;
            Y= b1*i + b2*j + b3*cote;

            // test d'impact
            test.x=X;
            test.y=Y;

            if(PtInRect(&zone_invalide,test))
// le point est-il invalide ?
                pDC->SetPixel(test,PALETTEINDEX(coul)); // 40*100=4000

        }

    // ------------------------------------------------------------
    // libérer GDI
    pDC->SelectPalette( pOldPal,true ) ;
    m_Palette.DeleteObject() ;
    pDC->SelectStockObject(ANSI_VAR_FONT);
    delete police;

}
```

Voici le résultat de l'exécution de ce programme :



3. Les ActiveX

Microsoft a conçu les composants COM (aussi appelés ActiveX) lorsque la plate-forme Windows s'est structurée en serveur d'application. Il s'agissait de créer des composants transactionnels capables de servir plusieurs applications, le système d'exploitation se chargeant de contrôler l'instanciation, la répartition de charge, la sécurité… Une de leurs caractéristiques intéressantes est leur implémentation en C++ - langage objet à l'exécution rapide - et un système d'interface écrit en IDL - interface description language - facilitant la mise à jour de l'implémentation.

Depuis une quinzaine d'années, il existe des millions de composants de ce type et Microsoft cherche à abandonner cette architecture au profit de .NET. Les ActiveX sont encore très utilisés dans les applications de type Visual Basic (non .NET) et paradoxalement par des applications Web qui trouvent là un moyen abordable d'augmenter le registre de fonctionnalités des navigateurs.

L'environnement .NET

1. Le code managé et la machine virtuelle CLR

Comme Java, .NET fait partie de la famille des environnements virtualisés. Le compilateur C++ managé ne produit pas du code assembleur directement exécuté par le microprocesseur mais un code intermédiaire, lui-même exécuté par une machine "virtuelle", la CLR - *Common Language Runtime*. Cette couche logicielle reproduit tous les composants d'un ordinateur - mémoire, processeur, entrées-sorties… Cette machine doit s'adapter au système d'exploitation qui l'héberge, et surtout optimiser l'exécution du code.

Il y a là une différence importante entre .NET et Java. Alors que Microsoft a évidemment fait le choix de se concentrer sur la plate-forme Windows, d'autres éditeurs ont porté Java sur leurs OS respectifs - Unix, AIX, Mac OS, Linux… La société Novell a cependant réussi le portage de .NET sur Linux, c'est le projet Mono.

Cet environnement virtualisé a une conséquence très importante sur les langages supportés ; les types de données et les mécanismes objets sont ceux de la CLR. Comme Microsoft était dans une démarche d'unification de ses environnements, plusieurs langages se sont retrouvés éligibles à la CLR, quitte à adapter un peu leur définition. Le projet de Sun était plutôt une création ex nihilo, et de ce fait seul le langage Java a réellement été promu sur la machine virtuelle JVM.

2. Les adaptations du langage C++ CLI

Le terme CLI signifie *Common Language Infrastructure* ; c'est l'ensemble des adaptations nécessaires au fonctionnement de C++ pour la plate-forme .NET / CLR.

a. La norme CTS

Le premier changement concerne les types de données. En successeur de C, le C++ standard a basé sa typologie sur le mot machine (**int**) et le caractère de 8 bits (**char**). Confronté à des problèmes de standardisation et d'interopérabilité, Microsoft a choisi d'unifier les types de données entre ses différents langages.

Les types primitifs, destinés à être employés comme variables locales (boucles, algorithmes…), sont des types "valeurs". Leur zone naturelle de stockage est la pile (stack), et leur définition appartient à l'espace de noms **System** :

wchar_t	**System::Char**	Caractère unicode
signed char	**System::SByte**	Octet signé
unsigned char	**System::Byte**	Octet non signé
double	**System::Double**	Décimal double précision
float	**System::Float**	Décimal simple précision
int	**System::Int32**	Entier 32 bits
long	**System::Int64**	Entier 64 bits
unsigned int	**System::UInt32**	Entier 32 bits non signé
unsigned long	**System::UInt64**	Entier 64 bits non signé
short	**System::Int16**	Entier court 16 bits
bool	**System::Boolean**	Booléen
void	**System::Void**	Procédure

Voici un exemple utilisant à la fois la syntaxe habituelle de C++ et la version "longue" pour définir des nombres entiers. Cette dernière ne sera utilisée qu'en cas d'ambiguïté, mais il s'agit en fait de la même chose.

```
int entier = 4; // alias de System::Int32
System::Int32 nombre = entier;
```

Les structures managées (**ref struct**) sont composées de champs reprenant les types ci-dessus. Elles sont créées sur la pile et ne sont pas héritables.

```
ref struct Point
{
public:
    int x,y;
} ;

int main(array<System::String ^> ^args)
{
    Point p; // initialisation automatique sur la pile : pas de gcnew
    p.x = 2;
    p.y = 4;

    return 0;
}
```

Comme tous les types valeurs, les structures managées n'autorisent pas l'effet de bord à moins qu'une référence explicite n'ait été passée (voir ci-dessous les références suivies).

Par opposition les classes managées (**ref class**) sont créées sur le tas et correspondent davantage au fonctionnement des classes (structure) du C/C++ standard. Évidemment, elles sont héritables, manipulées par références, et instanciées par l'opérateur **gcnew**.

```
ref class Personne
{
public:
    String^ nom;
```

```
    int age;
} ;

int main(array<System::String ^> ^args)
{
    Personne^ pers = gcnew Personne();
    pers->nom = "Marc";
    pers->age = 35;

    return 0;
}
```

Naturellement, les structures et les classes managées supportent la définition de méthodes, C++ est bien un langage objet !

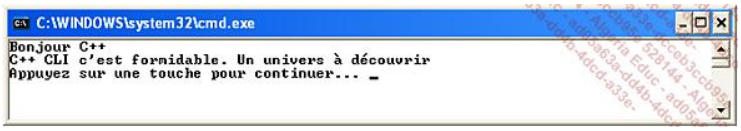
Les langages .NET sont fortement typés, en évitant autant que possible la généricité "fourre-tout" du **void*** issu du C, ou du **Variant** de VB. En opposition, la norme CTS prévoit que l'ensemble des types de données - valeurs ou référence - héritent d'un comportement commun **System::Object**. Nous découvrirons un peu plus loin comment cette caractéristique est exploitée dans les classes de collections d'objets.

b. La classe System::String

Parmi les types intrinsèques à .NET on trouve la classe **System::String** pour représenter les chaînes. Elle n'est pas aussi intégrée au langage C++ CLI que dans C#, mais cependant elle offre exactement les mêmes services. Microsoft l'a de plus aménagée pour faciliter la manipulation et la conversion avec **char*** :

Voici un premier exemple de construction de chaînes. On remarquera l'emploi facultatif du symbole L devant les littérales de chaîne, ainsi que le symbole ^ dont la signification sera expliquée un peu plus tard.

```
char* c = "Bonjour C++";
String ^ chaine1 = gcnew String(c); // construction à partir d'un char*
String ^ chaine2 = L"C++ CLI c'est formidable"; // Littérale de chaîne .NET
String ^ chaine3 = "Un univers à découvrir"; // idem

Console::WriteLine(chaine1);
Console::WriteLine(chaine2 + ". " + chaine3);
```



Nous poursuivons avec quelques exemples d'emploi de méthodes et de propriétés de la classe **System::String**. Cet extrait de code emploie une nouvelle instruction, **for each** qui réalise une itération sur l'ensemble des éléments d'une collection. Comme dans le cas de la STL, une chaîne est assimilée à une collection de caractères :

```
// manipuler les chaînes
String ^ prefixe = "Bon";
if(chaine1->StartsWith(prefixe))
Console::WriteLine("chaîne1 commence par {0}", prefixe);

// itérer sur les caractères avec une boucle for each
for each(wchar_t c in chaine1)
{
    Console::Write(" " + c);
}
Console::WriteLine();

// itérer classiquement avec un for
for(int i=0; i<chaine2->Length; i++)
    Console::Write( chaine2[i] );

Console::WriteLine();
```

La classe **System::String** contient de nombreuses méthodes et propriétés que tout programmeur a intérêt à découvrir. Beaucoup d'entre elles se retrouvent également dans la classe **string** de la STL.

Voici maintenant comment obtenir un **char*** à partir d'une **String**. L'opération repose sur le concept de marshalling, c'est-à-dire de mise en rang de l'information. Pourquoi est-ce si difficile ? Les **char*** n'ont pas beaucoup d'intérêt dans le monde .NET qui propose des mécanismes plus évolués. Mais ces mêmes **char*** sont indispensables dans le monde Win32 où un très grand nombre d'éléments de l'API les utilisent. Le passage d'un monde à l'autre, la transformation des données s'appelle le marshalling. C'est donc la classe **System::Runtime::InteropServices::Marshall** qui est chargée de "convertir" une **String** en **char*** :

```
// conversion vers char* (utile pour les accès natifs à Win32)
char* cetoile =
static_cast<char *> (Marshal::StringToHGlobalAnsi(chaine1).ToPointer());

// utilisation de la chaîne
// ...

// libérer le buffer
Marshal::FreeHGlobal(safe_cast<IntPtr>(cetoile));
```

Nous constatons que le programmeur est responsable d'allouer et de libérer le buffer contenant la chaîne exprimée en **char***. La présentation du garbage collector va donner des informations complémentaires à ce sujet.

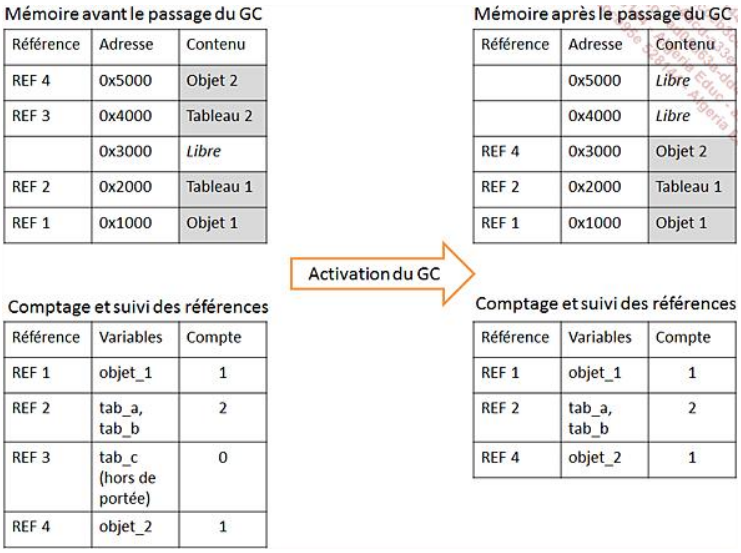
c. Le garbage collector

Une caractéristique des environnements virtualisés comme Java ou .NET est l'emploi d'un ramasse-miettes ou garbage collector en anglais. Il s'agit d'un dispositif de recyclage et de compactage automatique de la mémoire. Autant le fonctionnement de la pile est assez simple à systématiser, autant les algorithmes de gestion du tas diffèrent d'un programme à l'autre. Or, les langages comme le C et le C++ mettent en avant le mécanisme des pointeurs qui supposent une totale liberté dans l'adressage de la mémoire.

Quel est le principal bénéfice de la fonction garbage collector ? En ramassant les miettes, on évite le morcellement de la mémoire et de ce fait on optimise son emploi et sa disponibilité. Ceci est pourtant difficilement compatible avec l'emploi des pointeurs et leur arithmétique qui ne peut être prédite à la compilation. Les langages nouvellement créés pour les environnements virtualisés comme Java et C# ont donc fait "disparaître" la notion de pointeur au profit des seules références, puisqu'une référence n'est en principe pas "convertible" en adresse mémoire.

C# comme C++ CLI disposent malgré tout des pointeurs, mais leur utilisation est strictement encadrée pour ne pas perturber le fonctionnement du garbage collector.

Le ramasse-miettes est activé périodiquement par la CLR lorsque la mémoire vient à manquer ou qu'une allocation conséquente est sollicitée. Le programmeur peut également anticiper et demander lui-même son déclenchement. Le principe est de compter le nombre de références actives sur les zones mémoires allouées et de libérer toutes les zones qui ne sont plus référencées. Dans un second temps le garbage collector peut décider de déplacer des blocs mémoires pour compacter les espaces alloués et rendre disponibles de plus grands blocs. Les références correspondantes sont actualisées avec les nouvelles adresses, ce qui a pour conséquence de rendre les pointeurs inutilisables.



d. Construction et destruction d'objets

Le langage C++ CLI offre deux visages : le premier est une gestion "classique" de la mémoire sur un tas spécifique, avec constructeurs et destructeurs invoqués par les opérateurs **new** et **delete** (ou automatiquement si l'objet est instancié sur la pile). Le second repose sur la gestion "managée" de la mémoire, avec une utilisation de références sur le tas du CLR rendant possible l'usage du garbage collector.

C'est au moment de la définition de la classe que l'on opte pour l'un ou l'autre des modes. Si la classe est définie à l'aide du mot clé **ref**, il s'agit du mode managé. Autrement c'est le schéma classique.

Étudions la classe suivante :

```
// la présence du mot clé ref indique
// qu'il s'agit d'une classe managée
ref class Personne
{
public:

String ^ nom; // référence vers un objet managé de type String
int age;

    Personne()
    {
        nom = nullptr; // nullptr est une référence managée et non
                       // une adresse
        age = 0;
    }

    Personne(String ^ nom,int age)
    {
        // this est donc une référence managée et non un pointeur
        this->nom = nom;
        this->age = age;
    }
} ;
```

Le mot clé **nullptr** représente une référence managée et non une adresse. Dans l'univers classique **NULL** (littéralement **(void*)0**) représente une adresse qui ne peut être utilisée. Dans le cas des classes managées, c'est une constante dont le programmeur n'a pas à connaître la valeur puisqu'il n'y a pas d'arithmétique des pointeurs. Par commodité, **this** devient l'autoréférence de la classe et n'est pas un pointeur. Cependant Microsoft a choisi l'opérateur **->** pour accéder aux membres à partir d'une référence managée afin de ne pas troubler les programmeurs expérimentés dans le C++ classique.

Nous aurons également remarqué l'utilisation du symbole ***caret*** ^ qui désigne la référence vers un objet managé. On parle aussi de handle d'objet à cet effet. Voici à présent comment on procède pour instancier une classe managée :

```
int main(array<System::String ^> ^args)
{
```

```
    // déclaration d'une référence
    Personne ^ objet1; // une référence vers un objet managé

    // instanciation et invocation du constructeur
    objet1 = gcnew Personne("Corinne", 25);

    // utilisation de l'objet
    Console::WriteLine("objet1: nom={0} age={1}",
        objet1->nom, objet1->age);

    // suppression explicite de la référence
    objet1 = nullptr;

    return 0;
}
```

En fait dans notre cas la ligne **objet1 = nullptr** est facultative, car la variable objet1 est automatiquement détruite en quittant la fonction **main()**. Le garbage collector est systématiquement avisé que le compte de référence associée à objet1 doit être diminué d'une unité. Si ce compte devient nul, alors le garbage collector sait qu'il peut détruire l'objet et libérer la mémoire correspondante.

En conséquence de l'emploi du ramasse-miettes, le programmeur ne libère pas lui-même ses objets puisque c'est le dispositif automatique qui s'en charge. Il y a donc une différence fondamentale dans l'approche pour libérer les ressources demandées par un objet. Comme Java, les concepteurs de C++ CLI ont choisi le terme de finaliseur, sorte de destructeur asynchrone.

Le langage C++ CLI propose les deux syntaxes, le destructeur (noté **~**) et le finaliseur (noté **!**).

```
ref class Personne
{
public:

    String ^ nom;
    int age;

    ...

    ~Personne()
    {
        Console::WriteLine("Destructeur");
    }

    !Personne()
    {
        Console::WriteLine("Finaliseur");
    }
} ;
```

Le destructeur sera invoqué seulement si le programmeur détruit explicitement l'objet au moyen de l'instruction **delete**.

```
int main(array<System::String ^> ^args)
{
    // déclaration d'une référence
    Personne ^ objet1; // une référence vers un objet managé

    // instanciation et invocation du constructeur
    objet1 = gcnew Personne("Corinne", 25);

    // utilisation de l'objet
    Console::WriteLine("objet1: nom={0} age={1}",
        objet1->nom, objet1->age);

    // destruction explicite de l'objet -> seul le destructeur est appelé delete objet1;

    // suppression explicite de la référence
    objet1 = nullptr;

    return 0;
}
```

Le finaliseur sera invoqué si l'objet est détruit par le garbage collector (ce qui dans notre cas se produit au moment où la variable objet1 disparaît de la portée de la fonction) et à condition que le destructeur n'ait pas été invoqué.

```
int main(array<System::String ^> ^args)
{
    // déclaration d'une référence
    Personne ^ objet1; // une référence vers un objet managé

    // instanciation et invocation du constructeur
    objet1 = gcnew Personne("Corinne", 25);

    // utilisation de l'objet
    Console::WriteLine("objet1: nom={0} age={1}",
        objet1->nom, objet1->age);

    // pas destruction explicite de l'objet
    // delete objet1;

    // suppression explicite de la référence
    objet1 = nullptr;

    return 0;
}
```

e. La référence de suivi % et le handle ^

À présent, concentrons-nous sur les références de suivi % (tracking reference) et sur le handle ^ auquel nous allons élargir l'usage.

Nous commençons par quelques fonctions utilisant ces notations :

```
// le symbole % signifie la référence (alias)
void effet_de_bord_reference(int %nombre, int delta)
{
    nombre += delta;
}

// les handles sont comme des "pointeurs" pour les valeurs
int calcul_par_reference(int ^r1, int ^r2)
{
    int v1 = *r1; // déréférencement
    int v2 = *r2; // déréférencement

    return v1 + v2;
}
```

La référence de suivi % agit comme le symbole **&** en C++ classique. Elle crée un alias sur une variable de type valeur ou objet. Le handle ^ se comporte plutôt comme un pointeur en ce sens qu'il introduit une indirection qu'il convient d'évaluer (déréférencer).

Voyons maintenant comment sont utilisées ces fonctions :

```
// a est la référence d'un entier (et non l'entier lui-même)
int ^a = 10;

// le déréférencement * donne accès à la valeur
a = *a + 2;
effet_de_bord_reference(*a, 1); // il faut déréférencer x pour l'appel
Console::WriteLine("a = {0}", a);
```

Ce premier exemple augmente la valeur "pointée" par **x** jusqu'à 13. On remarquera l'analogie avec la notation "valeur pointée" * du C++ standard. La ligne la plus difficile est sans doute celle de la déclaration : **`int ^ a`** donne la référence d'un entier que l'on peut utiliser par **`*a`**. En C++ classique, l'instruction **`int& r=3`** n'est pas correcte, et de même que **`int* p`** n'alloue aucun entier.

Le second exemple est beaucoup plus proche de C++ classique :

```
// un entier créé sur la pile
int b = 20;

// une référence (alias) de la valeur b
int %c = b; // b et c désignent la même valeur

// modification "directe" de b
effet_de_bord_reference(b, 2);

// modification "directe" de c alias b
effet_de_bord_reference(c, 3);

Console::WriteLine("b = {0} / c = {1}", b, c);
```

En sortie de cet exemple, b et c représente la même valeur soit 20 + 2 + 3 = 25.

À l'appel d'une fonction exigeant un handle, le compilateur le fournit lui-même. Dans l'exemple qui suit, **a** est déclaré **`int ^`** alors que **b** est déclaré **`int`**.

```
int somme = calcul_par_reference(a, b);
Console::WriteLine("somme = {0} ", somme);
```

On aura remarqué dans l'implémentation de cette fonction qu'il convient de déréférencer chaque paramètre handle pour accéder à la valeur.

Considérons maintenant la classe complexe et deux fonctions :

```
ref class Complexe
{
public:
    double re,im;

    Complexe()
    {
        re = im = 0;
    }
} ;

// le handle vers un objet donne l'accès aux champs
```

```
void effet_de_bord_objet(Complexe ^ comp)
{
    comp->re = 5; // -> agit comme un déréférencement
    comp->im = 2;
}

// la référence sur un handle est comme un pointeur de pointeur
void effet_de_bord_objet_ref(Complexe ^% comp)
{
    comp = gcnew Complexe();
    comp->re=10;
    comp->im=20;
}
```

La première fonction reçoit comme paramètre un handle sur un **Complexe** ; c'est suffisant pour accéder directement aux champs de l'objet. Dans ce cas on peut considérer **->** comme un opérateur effectuant un déréférencement. En sortie de la fonction, les champs **re** et **im** égalent respectivement 5 et 2.

```
// modification par la fonction des champs de l'objet
effet_de_bord_objet(comp);
Console::WriteLine("re = {0} im ={1}",comp->re,comp->im);
```

Le paramètre de la seconde fonction est une référence de handle, sorte de pointeur de pointeur. Elle permet de substituer l'objet désigné comme paramètre par un autre :

```
// substitution par la fonction de l'objet référencé
effet_de_bord_objet_ref(comp);
Console::WriteLine("re = {0} im ={1}",comp->re,comp->im);
```

Bien qu'il soit impossible de déterminer la valeur de la référence **comp** (l'adresse de l'objet), on pourrait admettre que celle-ci a été changée par la fonction puisque c'est un autre objet qui est associé à cette référence.

f. Le pointeur interne et les zones épinglées

De façon à limiter les modifications sur les algorithmes faisant appel aux pointeurs, Microsoft a intégré ce mécanisme dans le fonctionnement de la CLR. Un pointeur interne est l'équivalent strict d'un pointeur C++, mais sur une zone managée. On se rappelle toutefois que le GC a la faculté de déplacer des blocs mémoires, ce qui rend l'utilisation du pointeur interne un peu plus lente que les pointeurs natifs, puisque la CLR est chargée d'actualiser ces pointeurs.

La zone épinglée est une zone mémoire rendue inamovible aux yeux du GC. On gagne ainsi en rapidité puisque le pointeur épingle n'a pas à être actualisé, ceci aux dépens du morcellement de la mémoire.

Voici un premier exemple pour illustrer le fonctionnement du pointeur interne. Il s'agit d'initialiser un pointeur interne à partir de l'adresse d'un tableau de double. À titre de comparaison, nous rappelons que la syntaxe pour instancier un tableau natif est inchangée (déclaration **int* pi**). L'instanciation de tableaux managés est par contre assez différente puisqu'elle utilise l'opérateur **gcnew** et utilise la classe **System::Array**. L'initialisation du pointeur interne se pratique en demandant l'adresse du premier élément du tableau **&pd[0]**. Les opérations d'accès par pointeur suivent les mêmes règles qu'en C++ classique sauf qu'en arrière-plan le GC peut déplacer la mémoire sans affecter l'exécution du programme, même avec de l'arithmétique des pointeurs.

```
// un tableau de 50 entiers
// c'est un pointeur classique non managé
int*  pi = new int[50];

// un tableau de 10 double
// c'est une référence (handle) managée
array<double> ^ pd = gcnew array<double>(10);
pd[0] = 1;

// un pointeur interne sur le tableau de
 double interior_ptr<double> pid = &pd[0];
while(++pid < &pd[0] + pd->Length)
{
    *pid = *(pid-1) + 1; // c'est beau l'arithmétique
des pointeurs !
}

// vérification 1,2,3...10
for(int i=0; i< pd->Length; i++)
    Console::Write("{0} ", pd[i] );

Console::WriteLine();
```

Dans le cas du pointeur épingle, il faut procéder en plusieurs temps. Primo, on fixe la mémoire en initialisant un pointeur épingle à partir d'une adresse. S'il s'agit d'un champ, c'est tout l'objet qui se trouve fixé en mémoire. Secundo, un pointeur natif est instancié à partir du pointeur épingle et les opérations par pointeurs ont lieu. Tertio, la zone est à nouveau rendue au contrôle complet du GC en affectant le pointeur épingle à une autre référence ou bien à **nullptr**.

```
// déclarer un pointeur épingle fixe la mémoire correspondante
pin_ptr<double> pin = &pd[0];

// obtention d'un pointeur "natif" pendant la durée de la manipulation
double * manip = pin;

while(++manip < &pd[0] + pd->Length)
{
    *manip = *(manip-1) * 2; // c'est beau l'arithmétique
```

```
des pointeurs !
}

// vérification 1,2,4,8...
for(int i=0; i< pd->Length; i++)
   Console::Write("{0} ", pd[i] );

// libération de la zone : elle peut à nouveau bouger
pin = nullptr;
```

g. Les tableaux et les fonctions à nombre variable d'arguments

Les tableaux managés se déclarent et s'instancient à l'aide d'une syntaxe particulière. En effet, l'une des limitations des tableaux C++ classique reste leur très grande similitude avec les pointeurs, ce qui les rend parfois risqués d'emploi. La plate-forme .NET leur confère des propriétés très utiles comme la longueur (Length) ou l'énumération. Devenus indépendants des pointeurs, ils sont également des objets "comme les autres" que le GC peut déplacer au gré des compactages de la mémoire.

Voici un premier exemple dans lequel une chaîne est découpée autour de séparateurs espace ou point-virgule, et les mots affichés les uns après les autres au moyen d'une boucle **for each**.

```
// un tableau de char
array<wchar_t> ^ sep  = { ' ', ';' };

// un tableau de chaînes
String^ phrase = "Bienvenue dans C++ CLI";
array<String^> ^ tabs = phrase->Split(sep);

// parcourir le tableau
for each(String^ mot in tabs)
    Console::WriteLine(mot);
```

L'instanciation par **gcnew** que nous avions utilisé précédemment permet de préciser la taille du tableau, mais aussi ses dimensions. Ici une matrice de 3x3, en 2 dimensions donc :

```
// un tableau à dimension double
array<float,2> ^ matrice = gcnew array<float,2>(3,3);
matrice[0,0] = 1;
matrice[0,1] = 2;
matrice[0,2] = 3;

matrice[1,0] = 4;
matrice[1,1] = 5;
matrice[1,2] = 6;

matrice[2,0] = 7;
matrice[2,1] = 8;
matrice[2,2] = 9;

// affichage des dimensions
Console::WriteLine("Longueur de la matrice = {0} ", matrice->Length);
Console::WriteLine("Longueur de 1ère dim = {0} ", matrice->GetLength(0));
Console::WriteLine("Longueur de 2ème dim = {0} ", matrice->GetLength(1));
```

Par voie de conséquence, les fonctions à nombre variable d'arguments de C/C++ pouvaient être rendues plus sûres grâce à des tableaux d'objets préfixés par **...** et suivant le dernier paramètre formel de la fonction :

```
/// <summary>
/// journalise un événement (message paramétré {0}...)
/// </summary>
void log(String^ message, ... array<System::Object^> ^ p );
```

h. Les propriétés

Pas de plate-forme Microsoft sans propriété ni événement : ce sont des éléments indispensables à la programmation graphique tout d'abord mais bien plus, généralement, si l'on y regarde de plus près. Une propriété est un type particulier de membre qui s'apparente à un champ auquel on définit des accesseurs pour préciser son comportement :

- Lecture seule
- Écriture seule

- Formatage des valeurs en lecture ou en écriture
- Contrôle de cohérence en lecture ou en écriture
- Calcul d'une valeur en lecture ou en écriture…

Les applications des propriétés sont innombrables et la quasi-totalité de l'API .NET repose sur leur emploi plutôt que sur des champs publics.

```
ref class Utilisateur
{
protected:
    String^ login, ^ password;
    bool isLogged;

public:
    Utilisateur()
    {
        isLogged = false;
    }

    // une propriété en lecture / écriture
    property String^ Login
    {
        String^ get()
        {
            return login;
        }

        void set(String^ value)
        {
            login = value;
        }
    }

    // une propriété en écriture seule
    property String^ Password
    {
        void set(String^ value)
        {
            password = value;
        }
    }

    void Authenticate()
    {
        isLogged = (login=="..." && password=="xxx");
    }

    // une propriété en lecture seule
    property bool IsLogged
    {
        bool get()
        {
            return isLogged;
        }
    }

} ;

int main(array<System::String ^> ^args)
{
    Utilisateur^ user1 = gcnew Utilisateur();
    user1->Login = "abc";
    user1->Password = "098";
    user1->Authenticate();

    Console::WriteLine("Utilisateur {0}, authentifié {1}",
        user1->Login, user1->IsLogged);

    return 0;
}
```

i. Les délégués et les événements

Les délégués de C++ CLI succèdent aux pointeurs de fonction peu sûrs du langage C. Nous l'aurons compris, tous les éléments faiblement typés - et donc potentiellement dangereux - du monde classique ont été revisités en .NET. Les délégués de C++ CLI ont les mêmes rôles que les pointeurs de fonctions : ils servent à définir des fonctions génériques, des fonctions auxiliaires…

L'exemple ci-dessous montre comment utiliser un délégué pour définir une fonction générique de comparaison entre 2 nombres :

```
// un délégué est un type de fonction
delegate int del_compare(int a,int b);

// voici une fonction qui respecte la signature du délégué
int comparer_entier(int n1,int n2)
{
    return n1-n2;
}

// cette fonction emploie un délégué
int calcule_max(int a,int b,del_compare ^ f_comp)
{
    if(f_comp(a,b)>0)
```

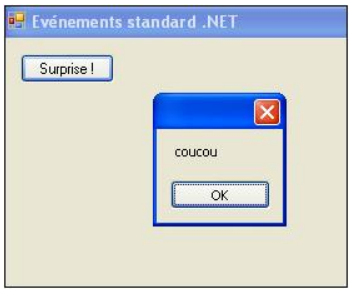
```
    // appel de la fonction au travers du délégué
        return a;
    else
        return b;
}

int main(array<System::String ^> ^args)
{
    // déclaration et instanciation du délégué
    del_compare ^ pcomp;
    pcomp = gcnew del_compare(&comparer_entier);

    // utilisation du délégué
    int max = calcule_max(34,22,pcomp);
    Console::WriteLine("Max = {0}",max);
    return 0;
}
```

Le rôle des délégués ne se limite pas à l'algorithmie puisqu'ils sont également employés pour démarrer des traitements sur des threads séparés et à l'implémentation des événements.

Un événement est un signal déclenché par un objet (ou une classe) qui peut être capté (on dit géré) par une ou plusieurs méthodes. L'exemple habituel est celui de l'événement **`Click`** sur un objet **`Button`** :

```
this->button1->Click += gcnew System::EventHandler(this, &Form1::button1_Click);
```

La méthode **`button1_Click`** est chargée d'afficher un message à l'écran :



Dans cet exemple, le délégué standard .NET est **`System::EventHandler`**. Mais nous verrons dans l'exemple du tableur comment déclarer ses propres signatures d'événements au moyen de délégués spécifiques.

j. Les méthodes virtuelles

Le fonctionnement habituel de C++, langage fortement typé avec phase d'édition des liens est celui de méthodes non virtuelles en cas d'héritage de classes. Toutefois c'est le principe inverse qui s'est imposé au fil des ans et de la montée en puissance des applications graphiques. Le programmeur C++ CLI doit donc être très assez précis quant à la mise en œuvre du polymorphisme. Le fait de déclarer une méthode comme étant **`virtual`** exigera des surcharges au niveau des sous-classes, qu'elles précisent **`override`** ou **`new`**. Dans le premier cas, la méthode reste virtuelle et le polymorphisme est appliqué en déterminant dynamiquement le type de l'objet. Dans le second cas, le polymorphisme est rompu et c'est la déclaration de l'objet qui prime.

Voici un premier exemple utilisant **`override`** :

```
ref class Compte
{
protected:
    double solde;
    int num_compte;
public:
    Compte(int num_compte)
    {
        this->num_compte = num_compte;
        solde = 0;
    }

    virtual void Crediter(double montant)
    {
         if(montant>0)
             solde += montant;
         else
             throw gcnew Exception("Erreur");
    }

    void Afficher()
    {
        Console::WriteLine("Compte n°{0} Solde = {1}",num_compte,solde);
    }

};

ref class Livret : public Compte
{
protected:
    double taux;
public:
```

```
    Livret(int num_compte) : Compte(num_compte)
    {
        taux = 0.0225;
    }

    virtual void Crediter(double montant) override
    {
        if(montant>0)
            solde += montant*(1+taux);
        else
            throw gcnew Exception("Erreur");
    }
} ;

int main(array<System::String ^> ^args)
{
    Compte^ c1 = gcnew Compte(1);
    c1->Crediter(100);
    c1->Afficher();

    Livret^ l2 = gcnew Livret(2);
    Compte ^ c2 = l2;
    c2->Crediter(100);
    c2->Afficher();

    return 0;
}
```

Dans cet exemple, l'objet **c2** désigne en fait un **Livret**, donc le compte est crédité avec intérêt :

En remplaçant **override** par **new**, la CLR aurait suivi la déclaration de **c2**, soit **Compte** et non **Livret**, et le compte n'aurait pas perçu d'intérêts.

k. Les classes abstraites et les interfaces

Par analogie avec les méthodes virtuelles pures de C++ classique, le langage C++ CLI propose les méthodes **abstraites**. Il s'agit de méthodes dont la définition n'a pas été donnée et qui doivent être implémentées dans des sous-classes concrètes avant d'être instanciables :

```
ref class Vehicule abstract
{
public:
    virtual void Avancer() abstract;
};

ref class Voiture : public Vehicule
{
public:
    virtual void Avancer() override
    {
        Console::WriteLine("Vroum");
    }
};
```

Nous mesurons la très grande proximité avec les méthodes virtuelles pures de C++, mais aussi le respect des mécanismes objets de .NET. Le mot clé **abstract** est plus adapté que le pointeur NULL de C++, tandis que le mot clé **override** souligne bien l'emploi du polymorphisme.

Le langage C++ CLI connaît aussi les interfaces, lesquelles s'apparentent à des classes totalement abstraites, mais sont surtout un moyen pour .NET de contourner l'héritage multiple au sein du framework.

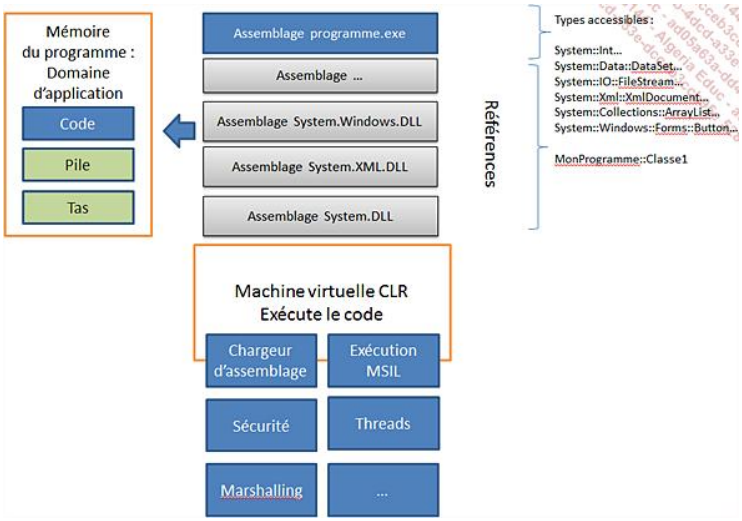
Les interfaces de classes se définissent sans recourir aux superclasses puisqu'elles sont totalement abstraites ; c'est donc la forme **new** qui prévaut dans les classes d'implémentation :

```
interface class Icompte
{
    void Crediter();
    void Afficher();
};


ref class Compte : Icompte
{
public:
    virtual void Crediter() new
    {
        //...
    }

    virtual void Afficher() new
    {
        //...
    }
};
```

3. Le framework .NET

Microsoft propose un framework très complet pour supporter le développement de toutes ses applications. .NET Framework est l'implémentation de la CLR pour Windows accompagné d'un immense réseau de classes organisées en namespaces et en assemblages. Tous les thèmes de la programmation sont abordés : réseau, graphisme, entrées-sorties, internet, sécurité, XML, programmation système… Nous n'en donnerons qu'un aperçu pour aider le programmeur à démarrer avec cet environnement.

Un assemblage est une DLL ou un EXE qui contient des types - classes, énumérations, délégués, structures… - répertoriés dans des espaces de noms.

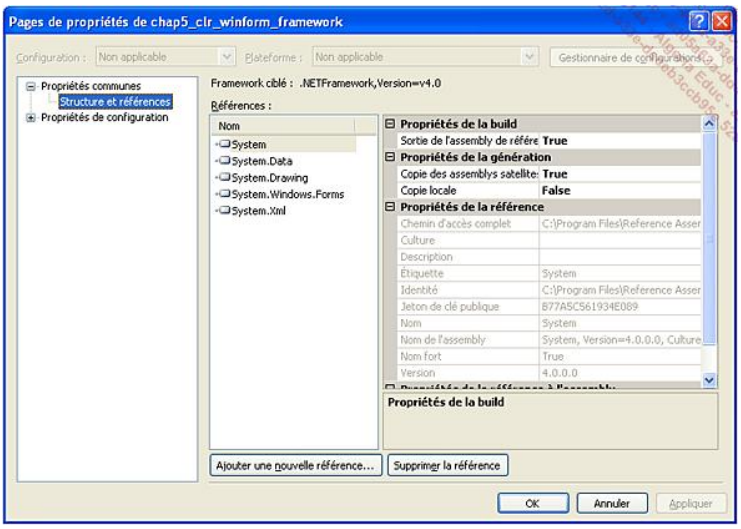


a. Les références d'assemblages

Pour accéder aux classes du framework, le programme doit d'abord référencer les assemblages qui les exposent, puis facultativement inclure les namespaces au moyen de l'instruction **`using namespace`**.

C'est au moyen de la commande **Projet** - **Propriétés** que l'on modifie la liste des références accessibles.

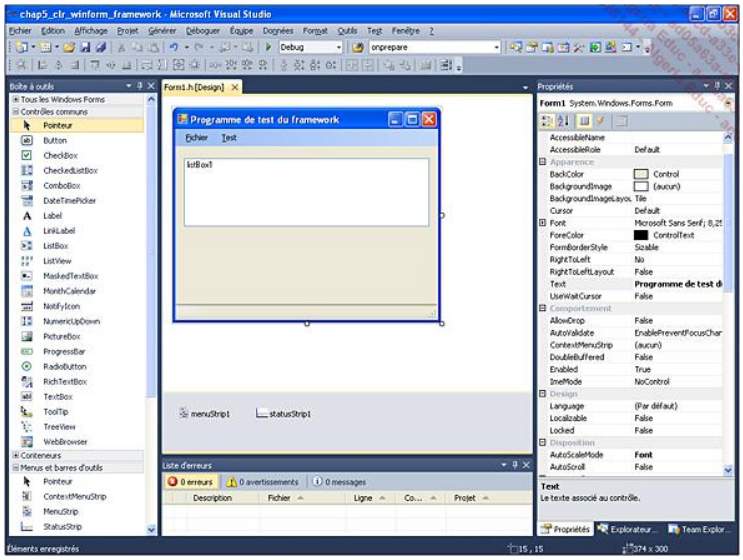
Dans notre exemple ci-dessous, il y a déjà 5 assemblages référencés :



b. L'espace de noms System::Windows::Forms

Cet espace correspond à la programmation d'applications graphiques (fenêtrées) sous Windows. Il définit les fenêtres, leur style, les contrôles - boutons, listes, cases à cocher…

Visual C++ propose un concepteur visuel d'écrans générant du code C++ :



Voici un extrait du code généré par Visual C++ :

```
#pragma once

namespace chap5_clr_winform_framework {

    using namespace System;
    using namespace System::ComponentModel;
    using namespace System::Collections;
    using namespace System::Windows::Forms;
    using namespace System::Data;
    using namespace System::Drawing;

    /// <summary>
    /// Description résumée de Form1
    /// </summary>
    public ref class Form1 : public System::Windows::Forms::Form
    {

    private: System::Windows::Forms::MenuStrip^  menuStrip1;
    protected:
    private: System::Windows::Forms::ToolStripMenuItem^  fichierToolStripMenuItem;
    private: System::Windows::Forms::ToolStripMenuItem^  nouveauToolStripMenuItem;
    private: System::Windows::Forms::ToolStripSeparator^  toolStripMenuItem1;

    private: System::Windows::Forms::StatusStrip^  statusStrip1;
    private: System::Windows::Forms::ListBox^  listBox1;

    private:
        /// <summary>
        /// Variable nécessaire au concepteur.
        /// </summary>
        System::ComponentModel::Container ^components;

#pragma region Windows Form Designer generated code
        /// <summary>
        /// Méthode requise pour la prise en charge du concepteur
        /// - ne modifiez pas
        /// le contenu de cette méthode avec l'éditeur de code.
        /// </summary>
        void InitializeComponent(void)
        {
            this->menuStrip1 = (gcnew System::Windows::Forms::MenuStrip());
            this->fichierToolStripMenuItem = (gcnew System::Windows::Forms::ToolStripMenuItem());
            this->nouveauToolStripMenuItem = (gcnew System::Windows::Forms::ToolStripMenuItem());
            this->toolStripMenuItem1 = (gcnew System::Windows::Forms::ToolStripSeparator());
            ...
            this->listBox1 = (gcnew System::Windows::Forms::ListBox());
            this->menuStrip1->SuspendLayout();
            this->SuspendLayout();
            //
            // menuStrip1
            //
            this->menuStrip1->Items->AddRange(gcnew cli::array<
 System::Windows::Forms::ToolStripItem^  >(2) {this-
>fichierToolStripMenuItem,
                this->testToolStripMenuItem});
            this->menuStrip1->Location = System::Drawing::Point(0, 0);
            this->menuStrip1->Name = L"menuStrip1";
            this->menuStrip1->Size = System::Drawing::Size(366, 24);
            this->menuStrip1->TabIndex = 0;
            this->menuStrip1->Text = L"menuStrip1";
            //
            // fichierToolStripMenuItem
            //
            this->fichierToolStripMenuItem->DropDownItems-
>AddRange(gcnew cli::array<
System::Windows::Forms::ToolStripItem^  >(6) {this->nouveauToolStripMenuItem,
            this->toolStripMenuItem1,
            this->ouvrirToolStripMenuItem,
            this->enregistrerToolStripMenuItem,
            this->toolStripMenuItem2,
```

```
            this->quitterToolStripMenuItem});
            this->fichierToolStripMenuItem->Name = L"fichierToolStripMenuItem";
            this->fichierToolStripMenuItem->Size = System::Drawing::Size(50, 20);
            this->fichierToolStripMenuItem->Text = L"&Fichier";
```

Lorsque l'on souhaite prendre en charge un événement particulier comme un clic sur un menu ou un bouton, c'est encore Visual C++ qui crée le code correspondant. Il n'y a plus qu'à implémenter le gestionnaire d'événement de manière appropriée :

```
private: System::Void
quitterToolStripMenuItem_Click(System::Object^  sender,
System::EventArgs^  e) {
    this->Close();
}
```

c. L'espace de noms System::IO

Cet espace de noms rénove totalement l'ancienne façon d'accéder aux fichiers héritée du langage C. Il y a maintenant plusieurs types de classes spécialisées :

FileInfo,**DirectoryInfo**	Opérations générales sur les fichiers et les répertoires - listes, déplacement…
FileStream	Flux fichier pour lire et écrire
Stream	Classe abstraite pour lire ou écrire (marche aussi avec le réseau)
StreamReader	Classe spécialisant un stream dans la lecture de caractères et de chaînes
StreamWriter	Classe spécialisant uns tream dans l'écriture de caractères et de chaînes
XmlTextReader	Lecture continue d'un flux XML (aussi appelée API SAX)
XmlTextWriter	Écriture au format XML
StringReader	Flux capable de lire dans une chaîne
StringWriter	Flux capable d'écrire dans une chaîne
MemoryStream	Flux mémoire

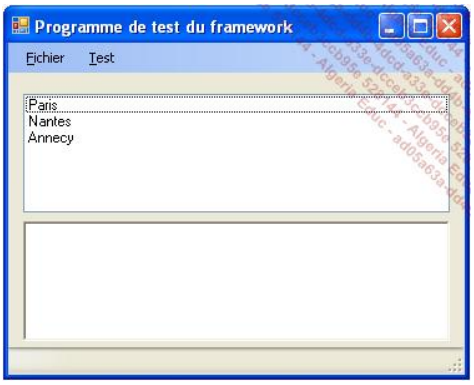
Voici un petit exemple d'écriture dans un fichier :

```
private: System::Void
enregistrerToolStripMenuItem_Click(System::Object^  sender,
System::EventArgs^  e) {
    // utilise le contrôle de sélection de fichier
    if(saveFileDialog1->ShowDialog() == System::Windows::Forms::DialogResult::Cancel)
    return;

    // nom du fichier sélectionné
    String^ filename = saveFileDialog1->FileName;

    // ouverture d'un flux fichier en écriture
    FileStream ^ fs = gcnew FileStream(filename, FileMode :: Create, FileAccess::Write);

    // adaptation d'un flux texte sur le flux fichier
    StreamWriter ^ sw = gcnew StreamWriter(fs);

    // écriture d'une chaîne dans le flux
    sw->Write("Ecriture dans un fichier");
    sw->Flush(); // purge du flux texte bufferisé

    // fermeture des flux
    sw->Close();
    fs->Close();

}
```

d. L'espace de noms System::Xml

Cet espace de noms propose de très nombreuses classes pour manipuler des données XML à l'aide des API SAX, DOM, XSL, XPath, XML schémas… La prise en charge de XML est donc très complète et fait figure de référence. L'exemple qui suit montre comment charger un extrait XML et l'analyser :

```
System::Void xMLToolStripMenuItem_Click(System::Object^  sender, System::EventArgs^  e) {
    // un extrait XML
    String ^ s ="<villes><ville>Paris</ville><ville>Nantes
</ville><ville>Annecy</ville></villes>";

    // instanciation d'un objet DOM
    XmlDocument ^ doc = gcnew XmlDocument();

    // initialisation de l'arbre DOM avec l'extrait XML
    doc->LoadXml(s);

    // parcours de l'arbre
    XmlNode ^ n = doc->DocumentElement->FirstChild;
    while( n != nullptr)
    {
        String ^ v = n->InnerText;
        listBox1->Items->Add(v);

        // prochain noeud collatéral
        n = n->NextSibling;
    }
}
```



e. L'espace de noms System::Data

Cet espace de noms est constitué des éléments d'accès déconnecté aux données relationnelles : **DataSet**, **DataTable**, **DataView**, **DataRelation**, **DataRow**, **DataColumn**… L'exemple suivant instancie une table DataTable et l'associe à la source de données d'un composant **DataGridView** :

```
System::Void dataToolStripMenuItem_Click(System::Object^  sender, System::EventArgs^  e) {
    // création d'une table et définition de 2 colonnes
    DataTable ^ dt = gcnew DataTable();
    dt->Columns->Add("Utilisateur");
    dt->Columns->Add("Login");

    // une ligne de données
    DataRow ^ dr;

    // la ligne est instanciée selon le modèle de la table
    dr = dt->NewRow();
    dr["Utilisateur"] = "Jean Valjean"; // 1ère valeur
    dr["Login"] = "jval"; // 2ème valeur
    dt->Rows->Add(dr); // rattachement à la table

    // autre façon d'indexer les colonnes
    dr = dt->NewRow();
    dr[0] = "Cosette";
    dr[1] = "cosy";
    dt->Rows->Add(dr);

    // encore une ligne
    dr = dt->NewRow();
    dr[0] = "Thénardier";
    dr[1] = "bistr";
    dt->Rows->Add(dr);

    // "affichage"
    dataGridView1->DataSource = dt;
}
```

f. L'espace de noms System::Collections

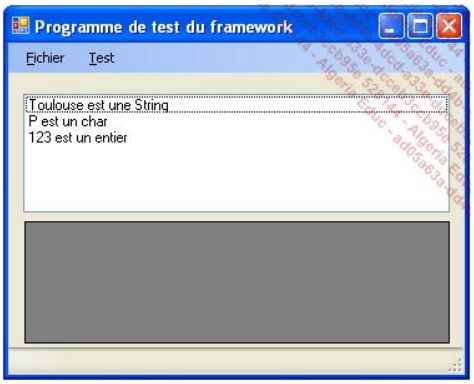
Les structures de données de l'espace de noms **System::Collections** ont fait partie de la première version de .NET Framework. Il s'agit de collections faiblement typées, qui exploitent pour beaucoup l'héritage de la classe **System::Object** par tous les types managés.

On retrouve dans cet espace de noms des piles, des files, des listes (tableaux dynamiques…). Voici justement un exemple d'emploi de la classe **ArrayList**. On découvre ici les avantages et les contraintes de l'approche faiblement typée. Il est très aisé d'attacher toutes sortes d'objets dans une liste, mais parfois plus délicat de retrouver leur type :

```
System::Void arrayListToolStripMenuItem_Click(System::Object^  sender, System::EventArgs^  e) {
    // un tableau liste contient des objets - valeurs ou références
    ArrayList ^ tab = gcnew ArrayList();

    tab->Add("Toulouse");        // ajout d'un objet référence
    tab->Add((wchar_t)'P');      // ajout d'un objet valeur
    tab->Add(123);               // ajout d'un objet valeur
```

```
    // créé 2 objets valeurs pour faciliter la détermination du type
    wchar_t c = 0;
    Object ^ ochar = c;

    int i=0;
    Object ^ oint = i;

    // énumération de la collection
    for each(Object ^ o in tab)
    {
     String ^ v;
     v = o->ToString(); // version textuelle de l'objet

     // teste le type de l'objet
     if(dynamic_cast<String ^>(o) != nullptr)
         v = v + " est une String";

     // pour les types valeurs dynamic_cast ne s'applique pas
     if(o->GetType()->Equals(ochar->GetType()))
         v = v + " est un char";

     if(o->GetType()->Equals(oint->GetType()))
         v = v + " est un entier";

     // affichage
     listBox1->Items->Add(v);
    }
}
```

L'exécution du programme indique bien que tous les types ont été identifiés :



Évidemment, si le type de la collection est homogène, il est plus simple d'utiliser systématiquement l'opérateur **dynamic_cast** voire l'opérateur de transtypage par coercition.

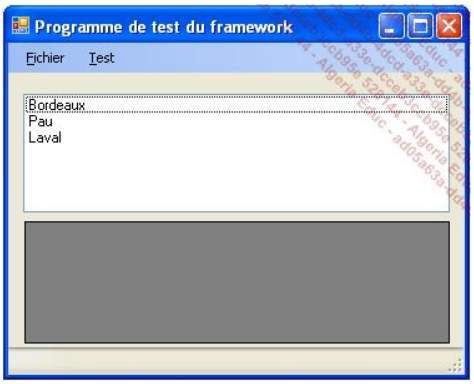
g. L'espace de noms System::Collections::Generic

Apparus avec la deuxième version de .NET Framework, les conteneurs génériques ont largement simplifié l'écriture des programmes puisqu'ils sont paramétrés - comme avec la STL - avec un type spécifique d'objet.

Comparer<T>	Classe de base pour implémenter des algorithmes de tris génériques
Dictionnary<T>	Table de hachage générique
LinkedList<T>	Liste générique doublement chaînée
List<T>	Liste générique
Queue<T>	File d'attente générique (aussi appelée pile FIFO)
SortedList<T>	Liste générique dont les éléments peuvent être triés
Stack<T>	Pile générique (aussi appelée pile LIFO)

L'exemple suit :

```
System::Void genericToolStripMenuItem_Click(System::Object^  sender, System::EventArgs^  e) {
    // cette liste ne contient que des String
    List<String^> ^ liste = gcnew List<String^>();

    // ajout d'objets d'après le type indiqué
    liste->Add("Bordeaux");
    liste->Add("Pau");
    liste->Add("Laval");

    // le parcours énumère les String
    for each(String^ s in liste)
        listBox1->Items->Add(s);
    }
```



h. Le portage de la STL pour le C++ CLI

Microsoft a débuté le portage de la STL sur .NET mais il semble que l'expérience ne sera pas menée à son terme. Il y a de grandes différences dans l'approche et dans la gestion des threads. À ce jour, la STL pour C++ CLI 2010 ne comporte que quelques classes alors que le framework .NET est d'ores et déjà très complet. Au-delà de la frustration, c'est sans doute certains programmes qui sont à porter (réécrire) sur cette nouvelle bibliothèque.

4. Les relations avec les autres langages : C#

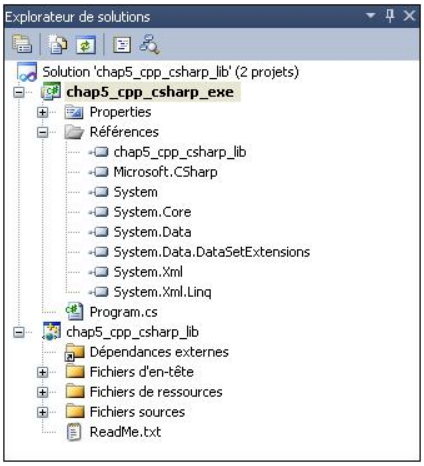
Notre démarche n'est pas ici de comparer C++ CLI et C#, mais plutôt d'illustrer les possibilités d'échanges et de partage entre les langages CLS - *Common Language Specification*.

Tous ces langages - C++ CLI, C#, VB.NET, F#... - partagent un socle commun, la CLR qui exécute un code intermédiaire MSIL indépendant. Chaque programmeur peut ainsi choisir son langage sans se préoccuper des problèmes d'interopérabilité qui avaient largement nui au développement d'applications pendant tant d'années. Le choix est alors plus une question d'aptitude, de préférence, de quantité de code existant, c'est beaucoup plus positif comme approche que de procéder par élimination.

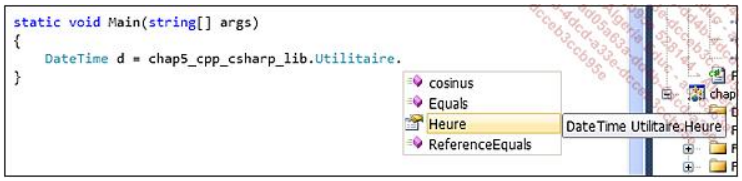
Nous proposons ici l'exemple d'une DLL écrite en C++ et d'une application graphique programmée en C#. Commençons par la DLL qui n'est constituée que d'une classe :

```
public ref class Utilitaire
{
public:
    static property DateTime Heure
    {
        DateTime get()
        {
            return DateTime::Now;
        }
    }

    static double cosinus(double a)
    {
        return 1 - a*a / 2;
    }
};
```

Après avoir créé un projet d'application console C#, nous ajoutons une référence à la DLL, ce qui est la condition pour que le programme C# puisse atteindre les types publics définis précédemment :



Le système d'aide à la saisie intellisense de Visual Studio retrouve bien les types et les méthodes publics :



Il n'y a plus qu'à tester :

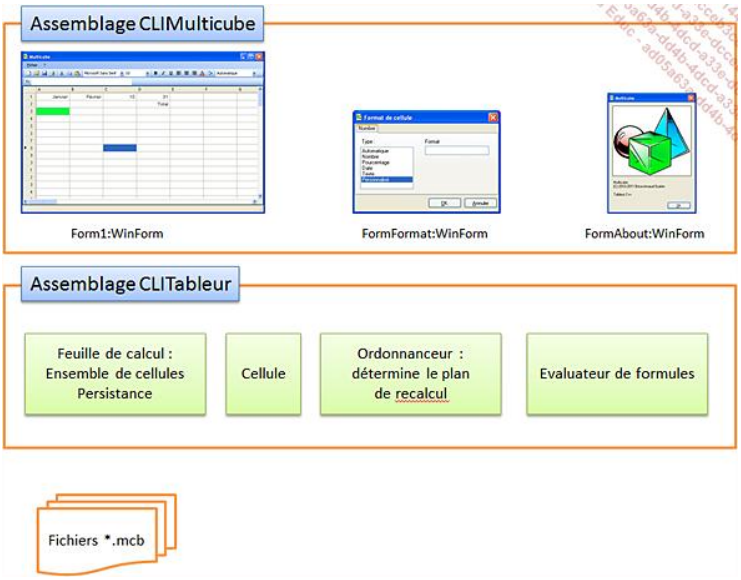
```
class Program
{
    static void Main(string[] args)
   {
        DateTime d = chap5_cpp_csharp_lib.Utilitaire.Heure;
        Console.WriteLine(d.ToLongTimeString());
    }
}
```

5. Une application en C++ pour .NET : le tableur

Nous proposons maintenant d'étudier la réalisation d'un tableur graphique en C++ CLI. Le but est ici d'expliquer la mise en œuvre de mécanismes de C++ CLI au travers d'une application conséquente, sans rentrer dans l'analyse fonctionnelle. Pour ceux qui souhaitent aller plus loin, l'ensemble du code source est documenté et fourni en téléchargement de fichiers complémentaires pour cet ouvrage.

a. Architecture du tableur

L'architecture repose sur deux assemblages distincts : l'interface graphique CLIMulticube et l'unité de calcul CLITableur. Le premier assemblage est un exécutable Winform tandis que le second est une bibliothèque de classes .DLL.



L'exécutable est principalement constitué de formulaires (d'écran) **`WinForm`** qui définissent l'apparence et le comportement de l'application graphique. On y trouve le formulaire principal **`Form1`**, la boîte de dialogue de formatage de cellule **`FormFormat`** et la boîte de dialogue d'à propos **`FormAbout`**. La fonction principale **`main()`** instancie et affiche **`Form1`** qui contrôle l'exécution de l'application jusqu'à sa fermeture. La programmation est événementielle et c'est au travers de menus et de barre d'outils que l'essentiel des opérations sont déclenchées. Comme nous le verrons un peu plus loin, des événements particuliers sont également gérés au niveau de la grille de donnée pour recalculer la feuille lorsque cela est nécessaire.

Le second assemblage est indépendant de l'interface graphique. Pour éviter les interdépendances, nous avons utilisé des événements pour ne pas tomber dans le piège de l'interface graphique qui commande un calcul, lequel doit ensuite être affiché sur l'interface graphique. La classe **`Feuille`** est un tableau de **`Cellule`**, cette classe représentant une valeur, une formule, un type, un format d'affichage, un style… Dans un tableur, les formules ne peuvent pas toujours s'évaluer de gauche à droite et de haut en bas, un ordonnanceur est intégré à la feuille de calcul pour déterminer l'ordre adéquat dans lequel les opérations d'actualisation auront lieu. Il manque encore l'évaluateur de formules qui calcule des expressions à base de constantes, de fonctions intrinsèques (cosinus, somme, moyenne…) ou de références à des cellules.

b. La feuille de calcul

Avant d'expliciter la classe **`Feuille`**, commençons par la classe **`Cellule`**. Une cellule est une formule dont l'évaluation donne une valeur. Cette valeur est ensuite formatée et affichée sur la grille de donnée en respectant un certain style. L'extrait qui suit donne un aperçu de cette classe où plusieurs mécanismes propres à C++ CLI sont utilisés : propriétés, énumérations, ainsi que les commentaires /// destinés à fabriquer la documentation du projet au format XML.

```
/// <summary>
/// Cellule représente une valeur ou une formule formatée
/// </summary>
public ref class Cellule
```

```
{

private:
    /// <summary>
    /// Valeur "calculée" d'une cellule (membre privé)
    ///    </summary>
    String^ value;

    /// <summary>
    /// Formule (membre privé)
    ///    </summary>
    String^ formula;

    /// <summary>
    /// Style gras (membre privé)
    ///    </summary>
    bool isBold;

    /// <summary>
    /// Style italique (membre privé)
    ///    </summary>
    bool isItalic;

    /// <summary>
    /// Style souligné (membre privé)
    ///    </summary>
    bool isUnderlined;

    /// <summary>
    /// Style couleur de fond (membre privé)
    ///    </summary>
    Color backgroundColor;

    /// <summary>
    /// Style couleur d'avant plan (membre privé)
    ///    </summary>
    Color foreColor;

    /// <summary>
    /// Style police (membre privé)
    ///    </summary>
    String^ fontName;

    /// <summary>
    /// Style taille de la police (membre privé)
    ///    </summary>
    float fontSize;

    /// <summary>
    /// Style alignement (membre privé)
    ///    </summary>
    DataGridViewContentAlignment alignment;

    public:
    /// <summary>
    /// Formatages possible de la valeur
    ///    </summary>
    enum class FormatType
    {
    AUTOMATIQUE, NOMBRE, POURCENTAGE, DATE, TEXT, PERSONNALISE
    };


    private:
    /// <summary>
    /// Formatage de la valeur (membre privé)
    ///    </summary>
    FormatType format;

    /// <summary>
    /// Chaîne de formatage (membre privé)
    ///    </summary>
    String^ formatString;

    public:

    /// <summary>
    /// Indique si la valeur peut être reconnue comme nombre
    ///    </summary>
    bool IsNumber();


    /// <summary>
    /// Constructeur
    ///    </summary>
    Cellule();

    /// <summary>
    /// Sérialise la cellule au format XML
    ///    </summary>
    XmlNode^ ToXml(XmlDocument^ doc, int col, int row);

    /// <summary>
    /// Désérialise la cellule à partir d'un noeud XML
    ///    </summary>
    void FromXml(XmlNode^ n);
} ;
```

La classe **`Feuille`** contient un tableau à 2 dimensions représentant la matrice de **`Cellule`**. Là encore des mécanismes de C++ CLI sont employés, comme les délégués et les événements. Nous verrons un peu plus loin comment l'interface graphique s'y "connecte". Nous trouvons aussi des propriétés facilitant les translations d'un domaine à l'autre, par exemple, pour retrouver le nom court d'un fichier.

```
/// <summary>
```

```
/// Feuille de calcul
/// </summary>
public ref class Feuille
{
public:
    /// <summary>
    /// Fonction appelée quand une cellule est mise à jour
    ///    </summary>
    delegate void del_cellUpdated(int col, int row, String^ value);

    /// <summary>
    /// Déclenché lorsqu'une cellule est mise à jour (calculée)
    ///    </summary>
    event del_cellUpdated^ OnCellUpdated;

    /// <summary>
    /// Fonction appelée lorsque le style est modifié
    ///    </summary>
    delegate void del_cellStyleUpdated(int col, int row);

    /// <summary>
    /// Déclenché lorsque le style est modifié
    ///    </summary>
    event del_cellStyleUpdated^ OnCellStyleUpdated;

    // taille de la feuille

    /// <summary>
    /// Nombre de lignes max
    ///    </summary>
    const static int ROWS = 100;

    /// <summary>
    /// Nombre de colonnes max
    ///    </summary>
    const static int COLS = 52;

private:
    int rows;
    int cols;

    /// <summary>
    /// Cellules composant la feuille de calcul
    ///    </summary>
    array<Cellule^,2>^ cells;

public:
    /// <summary>
    /// Accès indexé aux cellules
    ///    </summary>
    Cellule^ Cells(int col,int row);

    /// <summary>
    /// Constructeur
    ///    </summary>
    Feuille();

private:
    /// <summary>
    /// Evalue le contenu d'une cellule
    ///    </summary>
    String^ evalue(Cellule^ cell, int col, int row);

    /// <summary>
    /// Appelé pour capturer la valeur d'une cellule
    ///    </summary>
    void ev_OnGetCellValue(String^ name, double & value);

    /// <summary>
    /// Identifie les références des cellules contenues dans une plage
    ///    </summary>
    array<Cellule^>^ GetRange(String^ range);

    /// <summary>
    /// Instance d'évaluateur
    ///    </summary>
     Evaluator^ ev;

    /// <summary>
    /// Erreurs
    ///    </summary>
    StringBuilder^ errors;

public:
    /// <summary>
    /// Recalcule la feuille après détermination du plan
    ///    </summary>
    void recalc();

private:
    /// <summary>
    /// Chemin et nom de fichier (membre privé)
    ///    </summary>
    String^ filename;

    /// <summary>
    /// Nom du document (membre privé)
    ///    </summary>
    String^ fileshortname;

    /// <summary>
    /// Indique si une modification a eu lieu depuis la dernière sauvegarde (membre privé)
    ///    </summary>
    bool isModified;
```

```
    /// <summary>
    /// Enregistre le document vers le fichier filename
    ///    </summary>
    void SaveToFilename();

    /// <summary>
    /// Produit un flux XSLX
    ///    </summary>
    String^ GetExcelStream();

public:
    /// <summary>
    /// Constructeur applicatif
    ///    </summary>
    void Init();

    /// <summary>
    /// Indique si le document a été modifié depuis la dernière sauvegarde
    ///    </summary>
    property bool IsModified
    {
        bool get()
        {
             return isModified;
        }
    }

    /// <summary>
    /// Obtient ou définit le nom/chemin du document
    ///    </summary>
    property String^ Filename
    {
        String^ get()
        {
            return filename;
        }

        void set(String^ value)
        {
            this->filename = value;

            FileInfo^ f = gcnew FileInfo(filename);
            this->fileshortname = f->Name;
        }
    }

    /// <summary>
    /// Obtient le nom du document
    ///    </summary>
    property String^ Fileshortname
    {
        String^ get()
        {
            return fileshortname;
        }
    }

    /// <summary>
    /// Enregistre le document
    ///    </summary>
    void Save();

    /// <summary>
    /// Enregistre sous un autre nom
    ///    </summary>
    void Save(String^ filename);


    /// <summary>
    /// Charge le document
    ///    </summary>
    void Load(String^ filename);


    /// <summary>
    /// Enregistre une copie au format XLSX
    ///    </summary>
    void SaveCopyExcel(String^ filename);
   };
```

Comme détail d'implémentation de la classe **`Feuille`**, nous avons choisi la méthode **`GetRange`** qui emploie des listes génériques :

```
/// <summary>
/// Identifie les références des cellules contenues dans une plage
///    </summary>
array<Cellule^>^ Feuille::GetRange(String^ range)
{
    array<String^>^ names = range->Split(gcnew array<wchar_t> { ':' });
    System::Collections::Generic::List<Cellule^>^ res =
        gcnew System::Collections::Generic::List<Cellule^>();

    Evaluator^ ev = gcnew Evaluator();
    int col1 = 0, col2 = 0;
    int row1 = 0, row2 = 0;

    ev->CellNameToColRow(names[0], col1, row1);
    ev->CellNameToColRow(names[1], col2, row2);

    for (int i = col1; i <= col2; i++)
        for (int j = row1; j <= row2; j++)
        {
            res->Add(cells[i, j]);
```

```
        }

        return res->ToArray();
}
```

La méthode **SaveToFilename** met en œuvre quant à elle l'API XML de .NET que nous avons étudié précédemment :

```
/// <summary>
/// Enregistre le document vers le fichier filename
///    </summary>
void Feuille::SaveToFilename()
{
    XmlDocument^ doc = gcnew XmlDocument();
    XmlNode^ root = doc->CreateElement("document");
    doc->AppendChild(root);
    for(int col=0; col<cols; col++)
        for (int row = 0; row < rows; row++)
        {
            if (cells[col, row]->Value != "" || cells[col,
row]->Formula != ""
            {
                XmlNode^ n = cells[col, row]->ToXml(doc,col,row);
                root->AppendChild(n);
            }
        }

        StringBuilder^ sb = gcnew StringBuilder();
        XmlTextWriter^ writer = gcnew XmlTextWriter(gcnew StringWriter(sb));
        writer->Formatting = Formatting::Indented;

        doc->WriteTo(writer);
        writer->Flush();

        FileStream^ fs = gcnew FileStream(filename,FileMode::Create,FileAccess::Write);
        StreamWriter^ sw = gcnew StreamWriter(fs);
        sw->Write(sb->ToString());
        sw->Flush();
        fs->Close();

        isModified = false;
}
```

c. L'ordonnanceur de calcul

L'ordonnanceur qui détermine dans quel ordre les formules des cellules doivent être évaluées, repose sur l'implémentation d'un graphe. Cette structure de donnée n'existant pas nativement, nous l'avons créée en C++. Ici il y a pas de mécanisme propre à C++ CLI si ce n'est l'emploi des structures de données de **System::Collection::Generic (List, Stack)**, ainsi que de structures non typées comme **Hashtable**.

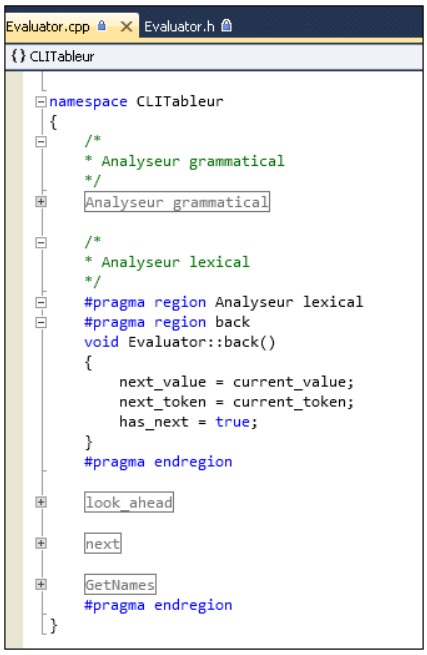
```
/// <summary>
/// Représente un sommet au sein d'un graphe
///    </summary>
public ref class Sommet
{
public:
    /// <summary>
    /// Nom du sommet
    ///    </summary>
    String^ name;

    /// <summary>
    /// Sommets du graphe adjacents
    ///    </summary>
    List<Sommet^>^ adjacents;

    public:
    Sommet()
    {
        adjacents = gcnew List<Sommet^>();
        name = "";
    }
};

/// <summary>
/// Représente un graphe dirigé
///    </summary>
public ref class Graphe
{
private:
    /// <summary>
    /// Ensemble des sommets du graphe (membre privé)
    ///    </summary>
    List<Sommet^>^ g;

public:
    /// <summary>
    /// Constructeur
    ///    </summary>
    Graphe()
    {
        g = gcnew List<Sommet^>();
    }

public:
    /// <summary>
    /// Ajoute un sommet au graphe, avec ses dépendances (sommets
adjacents)
    ///    </summary>
    void AjouterSommet(String^ name, array<String^>^ dependances)
    {
        Sommet^ adj;
```

```
        Sommet^ s;
        if ((s = GetSommetByName(name)) == nullptr)
        {
            // le sommet n'existe pas encore
            s = gcnew Sommet();
            s->name = name->ToUpper();
            g->Add(s);
        }

        // ajoute ses dépendances
        for (int i = 0; i < dependances->Length; i++)
            if ((adj = GetSommetByName(dependances[i])) != nullptr)
            {
                    s->adjacents->Add(adj);
            }
            else
            {
                // créé un nouveau sommet adjacent
                adj = gcnew Sommet();
                adj->name = dependances[i]->ToUpper();
                g->Add(adj);
                s->adjacents->Add(adj);
            }
        }

public:
    /// <summary>
    /// Identifie le sommet appelé name
    ///    </summary>
    Sommet^ GetSommetByName(String^ name)
    {
        for (int i=0;i<g->Count;i++)
        {
            Sommet^s = g[i];
            if (s->name == name)
                return s;
        }
        return nullptr;
    }


    /*
     * Tri topologique
     */
private:
    Hashtable^ visites;
    void init_visites()
    {
        visites = gcnew Hashtable();
    }

    System::Collections::Generic::Stack<Sommet^>^ pile;
    Sommet^ depart;

public:
    /// <summary>
    /// Effectue un tri topologique à partir des adjacences entre
sommet (dépendances inverses)
    ///    </summary>
    array<String^>^ TriTopologique()
    {
        pile = gcnew System::Collections::Generic::Stack<Sommet^>();
        List<String^>^ plan = gcnew List<String^>();
        hasError = false;
        errors = gcnew StringBuilder();

        for (int i=0;i<g->Count;i++)
        {
            Sommet^s = g[i];

            init_visites();
            depart = s;

            // le tri est basé sur un parcours en profondeur d'abord
            parcours_profondeur(s);
        }

        while (pile->Count != 0)
            plan->Add(pile->Pop()->name);

        return plan->ToArray();
    }

private:
    void parcours_profondeur(Sommet^ s)
    {
        if (s == nullptr)
            return;

        if (visites[s->name] != nullptr /*&& visites[s->name]
is bool*/)
            return;
        else
            visites[s->name] = true;

        pile->Push(s);
        for(int i=0;i<s->adjacents->Count;i++)
        {
            Sommet^ adj=s->adjacents[i];
            if (adj == depart)
            {
                // déjà visité : erreur, graphe cyclique
                hasError = true;
                errors->Append("Référence circulaire sur
```

```
" + depart->name + " : " + s->name + "\n");
                return;
            }

            parcours_profondeur(adj);
        }
    }

};
```

d. Zoom sur l'évaluateur

L'évaluateur étant une classe assez longue, nous avons recouru aux régions pour délimiter certaines parties du code :



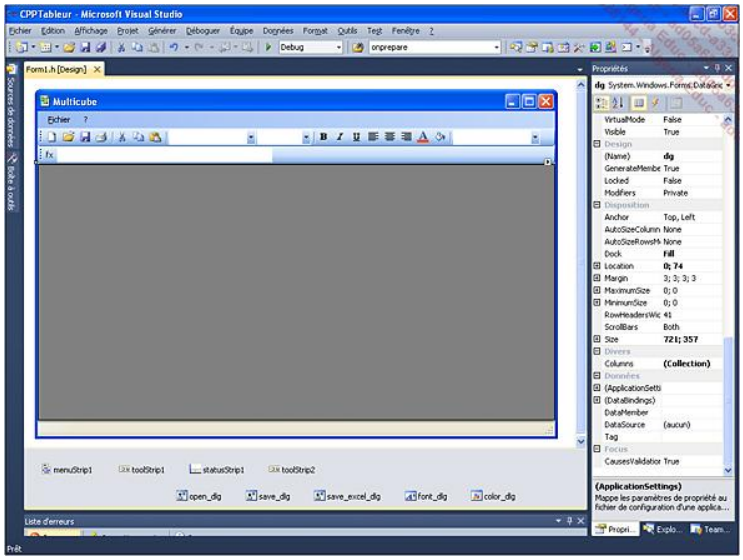
La méthode suivante est assez importante : elle utilise un événement pour solliciter une valeur issue d'une classe de niveau supérieur :

```
void Evaluator::cellule(String^ name, Variant^ f)
{
    double value = 0;

    // un événement cherche la valeur de la cellule sans ajouter une référence à Cells
    OnGetCellValue(name, value);
    f->nValue = value;
}
```

Sans événement, il aurait fallu transmettre une référence de **`Cells`** à l'objet **`Evaluator`** ce qui aurait créé une référence circulaire. Ces anomalies perturbent le fonctionnement du GC et provoquent des fuites mémoires (la mémoire est allouée et ne peut être relâchée).

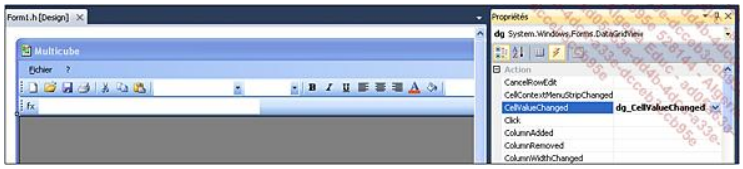
e. L'interface graphique

La présence d'un concepteur visuel pour C++ est quasiment indispensable s'il l'on souhaite créer des interfaces de haut niveau.



Ce concepteur laisse toute la liberté au programmeur pour positionner les contrôles sur un écran, modifier les propriétés d'apparence et de comportement, et enfin créer le code événementiel.

Découvrons le gestionnaire d'événement **`dg_CellValueChanged`** :



Cet événement est déclenché lorsque le contenu d'une cellule de la grille de donnée a évolué. La méthode détermine s'il s'agit d'une formule (commençant par = comme dans Excel) ou d'une valeur, puis demande le recalcul de la feuille :

```
#pragma region dg_CellValueChanged
void dg_CellValueChanged(Object^ sender,
DataGridViewCellEventArgs^ e)
{
    Object^ vo = dg->Rows[e->RowIndex]->Cells[e->ColumnIndex]->Value;
    String^ value = vo != nullptr ? vo->ToString() : "";

    if (value->StartsWith("="))
       sheet->Cells(e->ColumnIndex, e->RowIndex)->Formula = value;
   else
   {
        sheet->Cells(e->ColumnIndex, e->RowIndex)->FormattedValue = value;
   }

   // il faut recalculer puisque les valeurs ont changé
   sheet->recalc();
}
#pragma endregion
```

C'est la partie intéressante du programme du point de vue des événements : le recalcul va entraîner l'actualisation de certaines cellules. Il va d'abord éviter de passer une référence de la grille à la classe Feuille, voire à l'objet évaluateur, puis désactiver la détection de l'événement **`CellValueChanged`** car c'est le programme et non l'utilisateur qui actualise l'affichage.

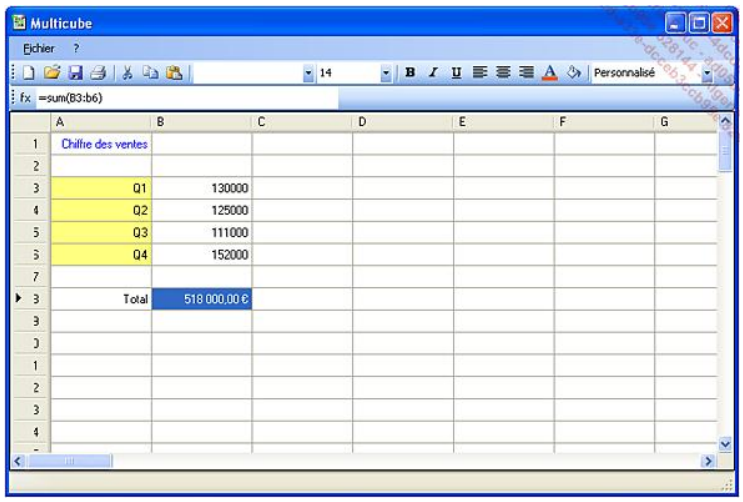
La méthode suivante contrôle justement la prise en charge de l'événement :

```
#pragma region SetCellValueHandler
void SetCellValueHandler(bool handle)
{
    if (handle)
        this->dg->CellValueChanged += gcnew
System::Windows::Forms::DataGridViewCellEventHandler(this,&Form1:
:dg_CellValueChanged);
    else
        this->dg->CellValueChanged -= gcnew
System::Windows::Forms::DataGridViewCellEventHandler(this,&Form1:
:dg_CellValueChanged);
}
#pragma endregion
```

Voici maintenant la définition de la méthode **`recalc()`** qui provoque l'actualisation des cellules sur la grille de données :

```
void Feuille::recalc()
{
    ev = gcnew Evaluator();
    errors = gcnew StringBuilder();

    ev->OnGetCellValue += gcnew Evaluator::del_get_cell_value(this,&Feuille::ev_OnGetCellValue);
```

```
    // création du plan
    Graphe^ graphe = gcnew Graphe();

    for (int i = 0; i < cols; i++)
        for (int j = 0; j < rows; j++)
        {
            if (cells[i, j]->Value->StartsWith("="))
            {
                cells[i, j]->Formula = cells[i, j]-> Value;
                graphe->AjouterSommet(
                    Evaluator::ColRowToName(i, j),
                    Evaluator::GetNames(cells[i, j]-> Formula));
            }
            else
            {
                if (cells[i, j]->Formula-> StartsWith("="))
                {
                    graphe->AjouterSommet(
                        Evaluator::ColRowToName(i, j),
                        Evaluator::GetNames(cells[i, j]-> Formula));
                }
            }
        }

        array<String^>^ plan = graphe->TriTopologique();
        errors->Append(graphe->getErrors());
        System::Collections::Generic::Stack<String^>^ pile =
            gcnew System::Collections::Generic::Stack<String^>();

        // exécution du plan
        for (int p = 0; p < plan->Length; p++)
        {
            int col = 0, row = 0;
            ev->CellNameToColRow(plan[p], col, row);

            if (cells[col, row]->Formula->StartsWith("="))
            {
                if (pile->Contains(plan[p]->ToUpper()))
                { // passer
                }
                else
                {
                    pile->Push(plan[p]->ToUpper());
                    cells[col, row]->Value = evalue(cells[col, row], col, row );

                    // déclenche un événement pour actualiser  l'affichage sur la grille                    OnCellUpdated(col, row, cells[col,  row]->Value);
                }
            }
        }
}
```



Dépasser ses programmes

1. Oublier les réflexes du langage C

Les programmeurs habitués au langage C ont sans doute prévu de nombreuses macros instructions traitées par le préprocesseur. C'est notamment le cas des constantes textuelles et de "fonctions" qui rendent bien des services.

Le langage C++ propose cependant des alternatives qui améliorent la sûreté du programme ainsi que sa rapidité.

Les constantes définies avec le mot clé **const**, tout d'abord, présentent l'énorme avantage d'être vérifiées par le compilateur et non par le préprocesseur. En cas d'erreur - visibilité, orthographe… - le compilateur peut donner beaucoup plus de contexte au programmeur.

Les fonctions en ligne (**inline**) permettent de développer des instructions sans construire tout un cadre de pile, avec les conséquences que cela peut avoir sur la transmission des arguments et sur les performances. Naturellement, leur mise en œuvre doit satisfaire à quelques contraintes mais il y a des bénéfices certains à les employer.

Considérons l'exemple suivant, dans lequel une constante symbolique est définie à l'attention du préprocesseur. En cas d'erreur d'écriture dans le **main**, le compilateur n'a aucun moyen de préciser dans quel fichier .h elle pourrait ne pas correspondre. De même nous avons décrit une macro qui compare deux entiers. Mais s'agissant d'une macro qui répète l'un de ses arguments, nous constaterons que des effets de bord apparaissent.

L'exemple propose une alternative à chaque problème, basée sur la syntaxe C++ plus sûre et plus rapide :

```
#define TAILLE 10
#define COMP(a,b) (a-b>0?1:(a-b==0?0:-1))

const int taille = 10;
inline int comp(int a,int b)
{
    if(a-b>0)
        return 1;
    else
    {
        if(a==b)
            return 0;
        else
            return -1;
    }
}

int _tmain(int argc, _TCHAR* argv[])
{
    // utilisation comme en langage C
    int* tableau = new int[TAILLE];
    int a = 2, b = 3;
    int z = COMP(a++, b); // quelle est la valeur de a en sortie ?
    printf("a = %d\n",a);

    // utilisation comme en C++
    A = 2;
    Z = comp(a++,b); // quelle est la valeur de a en sortie ?
    printf("a = %d\n",a);

    return 0;
}
```

À l'exécution, nous vérifions que l'utilisation de la macro produit un effet de bord indésirable sur la variable **a** :

2. Gestion de la mémoire

L'une des subtilités dont nous n'avons pas encore parlé est l'allocation puis la libération de tableaux d'objets. Considérons la classe **Point** dont le constructeur alloue de la mémoire pour représenter des coordonnées :

```
class Point
{
public:
    int* coordonnees;

    Point()
    {
        coordonnees = new int[2];
        printf("Point\n");
    }

    ~Point()
    {
        delete coordonnees;
        printf("~Point\n");
    }
} ;
```

Si nous procédons à l'instanciation d'un tableau de Point, il y aura autant d'appels au constructeur que d'objets créés :

```
Point* tab = new Point[3]; // trois constructeurs appelés car 3 points créés
```

Après utilisation, il est convenable de libérer ce tableau. Toutefois l'utilisation de **delete** sur cet objet **tab** peut avoir des comportements inattendus. Une partie des objets ne sera pas détruite, ou pire, une erreur d'exécution peut survenir. Il convient plutôt d'utiliser l'instruction **delete[]** :

```
//delete tab; // erreur, seul un destructeur est appelé
delete[] tab; // erreur, seul un destructeur est appelé
```

3. Concevoir des classes avec soin

Sans vouloir appauvrir les nombreuses normes élaborées pour C++, nous recommandons d'adopter un schéma type pour définir des classes en se concentrant sur le programme, mais sans oublier l'essentiel. Ce langage est à la fois très puissant, très riche, et recèle beaucoup de pièges dans ses constructions "non-dites".

La première chose à faire est de choisir des identificateurs qui sont explicites et parlants, et en même temps de respecter une convention de nommage. Mélanger **m_param**, **iParam**, **param**, tout appeler **champ1**, **champ2**,

champ3 sont des erreurs à éviter à tout prix.

Une classe est une entité complète, autonome, et le programmeur doit connaître à l'avance les membres qui la composent.

La seconde préoccupation est la visibilité de ces membres ; oublions les formules toutes faites "tous les champs sont privés et toutes les méthodes sont publiques". La réalité est toujours plus nuancée, et les méthodes d'accès - pour ne pas dire les propriétés - viennent rapidement contredire cette logique un peu simpliste.

La troisième recommandation est l'ordre dans lequel sont définis les membres. On peut commencer par les champs, puis les constructeurs, puis les méthodes publiques, enfin les méthodes non publiques. Il ne faut pas se priver de répéter pour chaque membre sa visibilité, ce qui évite des erreurs en cas de copier-coller intempestif.

Pour chaque méthode, on doit bien avoir à l'esprit si elle est statique, virtuelle, abstraite… Pour chaque champ, il faut déterminer son type, son nom, sa visibilité s'il est statique, constant…

La dernière recommandation est la documentation. Une bonne classe doit être accompagnée de beaucoup de commentaires. Ceux-ci ne doivent pas reformuler en moins bien ce qui est écrit en C++, mais plutôt développer des détails supplémentaires pour guider celui qui parcourt le code.

Comme C++ sépare la définition de l'implémentation, cela constitue une "optimisation" pour le programmeur. Le compilateur ne "moulinera" que les fichiers .cpp qui sont amenés à évoluer plus fréquemment que les fichiers .h, normalement plus stables car issus de l'étape de conception.

4. Y voir plus clair parmi les possibilités de l'héritage

Le langage C++ propose plusieurs modes d'héritage. Le plus utilisé, à juste titre, est l'héritage public. À quoi peuvent bien servir les autres modes, comme l'héritage privé ?

L'héritage public spécialise une classe. La classe de base exprime le concept le plus général, le plus abstrait, alors que les sous-classes vont plus dans le détail. Mais il existe toujours en principe une relation "est-un-cas-particulier-de" entre la classe dérivée et la classe de base ; le livret est un cas particulier de compte. La voiture est un cas particulier de véhicule…

De ce fait il faut se méfier des constructions syntaxiquement possibles mais sémantiquement insensées. La classe Chameau-Satellite n'a sans doute pas grande utilité. Une applet ne peut être aussi une horloge est un gestionnaire de souris. Ces constructions amènent de la complexité et produisent des programmes difficiles à maintenir. Faut-il en conclure que l'héritage multiple est à proscrire ? Sans doute pas. C'est un mécanisme indispensable mais qui doit être envisagé avec des précautions. C++ étant un langage général, il permet l'héritage multiple, il en a même eu besoin pour la STL. Mais le programmeur doit être prudent s'il est amené à recourir lui-même à cette construction. L'autre possibilité est de s'appuyer sur un framework qui prévoit l'héritage multiple dans un contexte encadré, sans risque.

Et l'héritage privé, finalement ? Là encore il s'agit d'un "truc" pour éviter l'explosion de code au sein d'un programme complexe. Tous les membres devenant privés, ce n'est pas pour créer des méthodes virtuelles que l'on recourt à ce mode. C'est pour repiquer du code sans trop s'embarrasser d'un quelconque polymorphisme.

De nos jours, il existe d'autres façons de contrôler la quantité de code, et la programmation par aspects AOP (*Aspect-Oriented Programming*) ouvre une voie prometteuse.

5. Analyser l'exécution d'un programme C++

Il existe des outils d'analyse de logiciels C++ (profilers) traquant les goulets de performances, les fuites mémoire et les opérations litigieuses. Sous Unix, l'utilitaire purify sert précisément à détecter les erreurs d'implémentations d'un programme C++.

Comme C++ est un langage assez proche de la machine, on peut également employer sous Windows l'utilitaire Process Viewer pour analyser le fonctionnement d'un programme.

La conception orientée objet

1. Relation entre la POO et la COO (Conception Orientée Objet)

La conception d'un programme orienté objet en C++ est une étape qui demande une certaine maturation. L'essence d'un tel programme ne tient pas seulement à l'implémentation des algorithmes qu'il contient, mais bien à la structure des classes et aux relations qui les réunissent. Nous nous proposons dans cette partie de décrire les approches de méthodes liées à la conception de programme C++.

a. L'approche initiale de C++

Avant tout, rappelons-nous que C++ a été créé pour créer des applications à destination du domaine des télécommunications. Ce langage n'est donc pas seulement un travail de recherche, mais c'est aussi le fruit d'un travail d'ingénierie informatique. Bien entendu, son concepteur Bjarne Stroustrup s'est assuré qu'il était suffisamment général pour être adapté à d'autres situations. Pour atteindre cet objectif, il a conçu trois éléments essentiels :

- 1. le langage C++ en lui-même.
- 2. la bibliothèque standard S.T.L.
- 3. une méthode de conception orientée objet pour C++.

La méthode proposée par Bjarne Stroustrup repose évidemment sur son expérience de conception d'applications antérieure à C++. Elle repose sur un certain nombre de thèmes forts, dont certains sont des classiques des méthodes de conception logicielle, orientée objet ou pas. Citons parmi ces thèmes la délimitation du système, la complexité des applications par rapport au degré d'abstraction des outils, ou encore les cycles de conception et de programmation perçus comme des activités itératives.

Le processus de développement minimal est constitué de trois étapes :

- 4. analyse
- 5. conception
- 6. implémentation

L'étape qui nous préoccupe est justement celle de la conception. L'auteur de la méthode organise des sous-processus à partir du découpage suivant :

- identification des classes, des concepts et de leurs relations les plus fondamentales ;
- spécification des opérations, c'est-à-dire des méthodes ;
- spécification des dépendances (des cardinalités dirait-on en UML) ;
- identification des interfaces pour les classes les plus importantes ;
- réorganisation de la hiérarchie des classes ;
- utilisation de modèles pour affiner le diagramme.

En conclusion, Bjarne Stroustrup a proposé une méthode générale, adaptée à la conception de programmes C++. Il est vraisemblable que le langage lui-même a été influencé par la méthode de conception orientée objet.

b. UML et C++

Un des intérêts de l'approche décrite ci-dessus vient du fait qu'elle est compatible avec un système de notation comme UML. De fait, UML a été créé à une époque où C++ était l'un des rares langages de programmation

orientée objet disponibles pour l'industrie.

UML est l'acronyme d'*Unified Modeling Language*. Il s'agit d'un formalisme qui unifie les travaux en matière de conception orientée objet développés pour la plupart au cours des années 70 et 80. Ce formalisme est à l'initiative d'une société, Rational, qui a ensuite proposé le logiciel de modélisation Rose, ainsi qu'une méthode à part entière appelée RUP (*Rational Unified Process*).

Comme C++ était l'un des principaux langages du marché, il a fortement influencé l'élaboration d'UML. Aussi, est-il fréquent de trouver des logiciels de modélisation UML capables de transformer automatiquement un diagramme de classes en programme C++, sans toutefois pouvoir fournir l'implémentation correspondante !

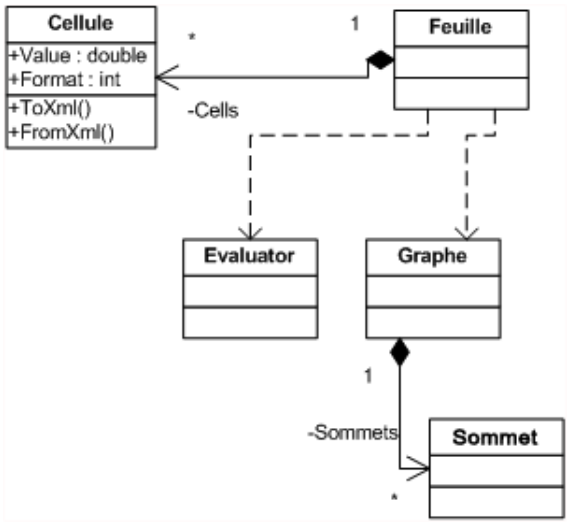
Le formalisme UML est constitué de neuf diagrammes. Il n'est souvent pas nécessaire d'exécuter l'ensemble des diagrammes pour parvenir à une modélisation conforme. Par contre, certains diagrammes nécessitent de s'y prendre en plusieurs fois. On parle alors d'un processus de modélisation itératif.

Voici la liste des diagrammes constitutifs du formalisme UML 1 :

Diagramme des cas d'utilisations	Vision fonctionnelle	Pose les limites du système en listant les acteurs et leurs intentions, sans s'attarder sur le "comment".
Diagramme de collaboration	Vision fonctionnelle	Les cas d'utilisations sont décrits sous la forme de scénarii textuels avant d'être formalités en diagrammes de collaboration. C'est le début du comment.
Diagramme d'activité	Vision fonctionnelle	C'est une version "workflow" du diagramme de collaboration plus adaptée à la description de certains scénarii.
Diagramme état-transition	Vision dynamique	C'est une version technique d'un workflow ; on considère des changements d'état au sein de classes qui prennent progressivement forme.
Diagramme de séquence	Vision dynamique	Semblable au diagramme d'état transition, le diagramme de séquence étudie la constitution de classe sous l'angle du temps, en faisant progressivement ressortir des méthodes.
Diagramme du domaine	Vision "métier"	Ce diagramme ne fait pas partie d'UML mais il est indispensable. Les classes métiers sont souvent structurées à partir des tables SQL.
Diagramme de classes	Vision statique	Ce diagramme est un peu la version affinée, superposée des précédents diagrammes.
Diagramme de composants	Vision système	Lorsque les classes sont stéréotypées, pourvues d'interfaces techniques, elles sont appelées composants.
Diagramme de déploiement	Vision système	Ce diagramme précise sur quels nœuds les composants doivent être déployés.

Des outils tels que Visio Architect ou Enterprise Architect sont capables de reproduire le diagramme de classes d'un programme, ou inversement de générer du C++ à partir d'un diagramme de classes. Comme UML et C++ sont deux langages très généraux, ils s'entendent parfaitement et toute la panoplie du C++ peut être écrite en UML.

Le diagramme de classe suivant montre comment modéliser une partie du tableur **`CLIMulticube`** étudié au chapitre Les univers de C++. C'est bien la preuve que le formalisme UML n'est aucunement réservé aux langages C# ou Java.



2. Les design patterns

Les design patterns sont des façons standard d'aborder des problèmes logiciels. Il s'agit de constructions de classes répertoriées que l'on adapte (développe) au gré des besoins.

Nous avons vu au chapitre Les univers de C++ que les MFC implémentaient le pattern MVC (Modèle Vue Contrôleur) à travers l'architecture document vue. Il existe de très nombreux modèles et certains générateurs de code peuvent constituer l'ossature d'une application en paramétrant des patterns pour un langage donné.

Pour illustrer les patterns, nous développons l'exemple du Singleton. Il s'agit de veiller à ce qu'une classe ne puisse être instanciée qu'une et une seule fois. Un exemple habituel d'utilisation de ce pattern concerne les fichiers de configuration qui ne doivent pas exister en plusieurs exemplaires au sein d'une application.

```
class Configuration
{
private:
    static Configuration* instance;
    Configuration()
    {
        // Le constructeur privé empêche toute instanciation
externe à la classe
    }

public:
    static Configuration* getInstance()
    {
        if(instance==NULL)
        {
            printf("Instanciation de Configuration\n");
            instance = new Configuration();
        }

        return instance;
    }

public:
    bool param1,param2;
} ;

Configuration* Configuration::instance=NULL;

int _tmain(int argc, _TCHAR* argv[])
{
    // utilisation du singleton
    Configuration::getInstance()->param1 = true;
```

```
    Configuration::getInstance()->param2 = true;

    return 0;
}
```

Comme le constructeur est privé, il est impossible d'instancier la classe **Configuration** en dehors de celle-ci. Cependant, nous pouvons confier cette instanciation à une méthode publique et statique, **getInstance()**. Cette dernière s'appuie sur un champ statique, **instance**, pour contrôler l'unique instanciation de la classe. Au premier appel de **getInstance()**, le champ **instance** égale **NULL** et la classe est instanciée. Tous les appels successifs travailleront à partir de cet unique objet. De cette façon, les paramètres de la configuration **param1** et **param2** n'existeront qu'en un seul exemplaire au sein du programme.

La copie d'écran ci-dessous montre que l'instanciation n'a eu lieu qu'une seule fois :